

=====

文件夹: class032_AdvancedDataStructures

=====

[Markdown 文件]

=====

文件: ADDITIONAL_PROBLEMS.md

=====

补充题目列表

本文件记录了为 [class035] (file:///D:/Upan/src/algorith-journey/src/algorith-journey/src/class035) 文件夹中高级数据结构设计题目找到的补充题目，包括题目名称、来源、内容描述以及网址链接。

1. 设计有 setAll 功能的哈希表

题目描述

哈希表常见的三个操作是 put、get 和 containsKey，而且这三个操作的时间复杂度为 $O(1)$ 。现在想加一个 setAll 功能，就是把所有记录 value 都设成统一的值。请设计并实现这种有 setAll 功能的哈希表，并且 put、get、containsKey 和 setAll 四个操作的时间复杂度都为 $O(1)$ 。

来源

- 牛客网: [设计有 setAll 功能的哈希表] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>)

相关实现

- Java: [Code01_SetAllHashMap.java] (file:///D:/Upan/src/algorith-journey/src/algorith-journey/src/class035/Code01_SetAllHashMap.java)
- Python: [Code01_SetAllHashMap.py] (file:///D:/Upan/src/algorith-journey/src/algorith-journey/src/class035/Code01_SetAllHashMap.py)
- C++: [Code01_SetAllHashMap.cpp] (file:///D:/Upan/src/algorith-journey/src/algorith-journey/src/class035/Code01_SetAllHashMap.cpp)

2. LRU 缓存机制

题目描述

运用你所掌握的数据结构，设计和实现一个 LRU（最近最少使用）缓存机制。它应该支持以下操作：获取数据 get 和写入数据 put。

来源

- LeetCode 146: [LRU Cache] (<https://leetcode.com/problems/lru-cache/>)
- 剑指 Offer II 031: [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
- 牛客网: [设计 LRU 缓存结构] (<https://www.nowcoder.com/practice/5dfded165916435d9defb053c63f1e84>)

相关实现

- Java: [Code02_LRU. java] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code02_LRU. java)
- Python: [Code02_LRU. py] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code02_LRU. py)
- C++: [Code02_LRU. cpp] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code02_LRU. cpp)

3. O(1)时间插入、删除和获取随机元素

题目描述

设计一个支持在平均时间复杂度 $O(1)$ 下执行以下操作的数据结构:

1. insert(val): 当元素 val 不存在时返回 true，并向集合中插入该项，否则返回 false
2. remove(val): 元素 val 存在时，从集合中移除该项，返回 true，否则返回 false
3. getRandom: 随机返回现有集合中的一项，每个元素应该有相同的概率被返回

来源

- LeetCode 380: [常数时间插入、删除和获取随机元素] ([https://leetcode.com/problems insert-delete-getrandom-o1/](https://leetcode.com/problems	insert-delete-getrandom-o1/))
- 牛客网: [O(1)时间插入、删除和获取随机元素] (<https://www.nowcoder.com/discuss/353149939293298688>)

相关实现

- Java: [Code03_InsertDeleteRandom. java] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code03_InsertDeleteRandom. java)
- Python: [Code03_InsertDeleteRandom. py] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code03_InsertDeleteRandom. py)
- C++: [Code03_InsertDeleteRandom. cpp] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code03_InsertDeleteRandom. cpp)

4. 允许重复元素的 O(1) 数据结构

题目描述

设计一个支持在平均时间复杂度 $O(1)$ 下执行以下操作的数据结构（允许重复元素）:

1. insert(val): 将一个元素 val 插入到集合中，返回 true
2. remove(val): 如果元素 val 存在，则从中删除一个实例，返回 true，否则返回 false
3. getRandom: 随机返回集合中的一个元素，每个元素被返回的概率与其在集合中的数量成线性关系

来源

- LeetCode 381: [常数时间插入、删除和获取随机元素-允许重复] ([https://leetcode.com/problems insert-delete-getrandom-o1-duplicates-allowed/](https://leetcode.com/problems	insert-delete-getrandom-o1-duplicates-allowed/))
- LeetCode 380: [常数时间插入、删除和获取随机元素] ([https://leetcode.com/problems insert-delete-getrandom-o1/](https://leetcode.com/problems	insert-delete-getrandom-o1/))

相关实现

- Java: [Code04_InsertDeleteRandomDuplicatesAllowed. java] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code04_InsertDeleteRandomDuplicatesAllowed. java)
- Python: [Code04_InsertDeleteRandomDuplicatesAllowed. py] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code04_InsertDeleteRandomDuplicatesAllowed. py)
- C++: [Code04_InsertDeleteRandomDuplicatesAllowed. cpp] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code04_InsertDeleteRandomDuplicatesAllowed. cpp)

5. 数据流的中位数

题目描述

设计一个支持以下两种操作的数据结构：

1. void addNum(int num) – 从数据流中添加一个整数到数据结构中
2. double findMedian() – 返回目前所有元素的中位数

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

来源

- LeetCode 295: [数据流的中位数] (<https://leetcode.com/problems/find-median-from-data-stream/>)
- 剑指 Offer 41: [数据流中的中位数] (<https://leetcode.cn/problems/shu-ju-liu-zhong-de-zhong-wei-shu-lcof/>)
- LeetCode 480: [滑动窗口中位数] (<https://leetcode.com/problems/sliding-window-median/>)

相关实现

- Java: [Code05_MedianFinder. java] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code05_MedianFinder. java)
- Python: [Code05_MedianFinder. py] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code05_MedianFinder. py)
- C++: [Code05_MedianFinder. cpp] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code05_MedianFinder. cpp)

6. 最大频率栈

题目描述

实现一个类似栈的数据结构，支持以下操作：

1. push(val): 将一个整数 val 压入栈顶
2. pop(): 删除并返回栈中出现频率最高的元素

如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素

来源

- LeetCode 895: [最大频率栈] (<https://leetcode.com/problems/maximum-frequency-stack/>)
- 牛客网: [最大频率栈] (<https://www.nowcoder.com/discuss/791601453080055808>)

相关实现

- Java: [Code06_MaximumFrequencyStack. java] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code06_MaximumFrequencyStack. java)
- Python: [Code06_MaximumFrequencyStack. py] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code06_MaximumFrequencyStack. py)
- C++: [Code06_MaximumFrequencyStack. cpp] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code06_MaximumFrequencyStack. cpp)

7. 全 O(1) 的数据结构

题目描述

设计一个数据结构支持以下操作，所有操作的时间复杂度都为 O(1)：

1. inc(key)：将 key 的计数增加 1，如果 key 不存在则插入计数为 1 的 key
2. dec(key)：将 key 的计数减少 1，如果计数变为 0 则删除 key
3. getMaxKey()：返回计数最大的任意一个 key，如果不存在返回空字符串
4. getMinKey()：返回计数最小的任意一个 key，如果不存在返回空字符串

来源

- LeetCode 432: [全 O(1) 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>)
- LeetCode 146: [LRU 缓存] (<https://leetcode.com/problems/lru-cache/>)

相关实现

- Java: [Code07_A1101. java] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code07_A1101. java)
- Python: [Code07_A1101. py] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code07_A1101. py)
- C++: [Code07_A1101. cpp] (file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class035/Code07_A1101. cpp)

其他相关题目

8. LFU 缓存

- LeetCode 460: [LFU Cache] (<https://leetcode.com/problems/lfu-cache/>)

9. 设计哈希映射

- LeetCode 706: [设计哈希映射] (<https://leetcode.com/problems/design-hashmap/>)

10. 设计哈希集合

- LeetCode 705: [设计哈希集合] (<https://leetcode.com/problems/design-hashset/>)

11. 设计停车系统

- 力扣 1603: [设计停车系统] (<https://leetcode.cn/problems/design-parking-system/>)

12. 每隔 n 个顾客打折

- 力扣 1357: [每隔 n 个顾客打折] (<https://leetcode.cn/problems/apply-discount-every-n-orders/>)

13. 设计自助结算系统

- LCR 184: [设计自助结算系统] (<https://leetcode.cn/problems/dui-lie-de-zhui-da-zhi-1cof/solutions/>)

14. 滑动窗口中位数

- LeetCode 480: [滑动窗口中位数] (<https://leetcode.com/problems/sliding-window-median/>)

15. 设计链表

- LeetCode 707: [设计链表] (<https://leetcode.com/problems/design-linked-list/>)

16. 设计循环队列

- LeetCode 622: [设计循环队列] (<https://leetcode.com/problems/design-circular-queue/>)

17. 设计循环双端队列

- LeetCode 641: [设计循环双端队列] (<https://leetcode.com/problems/design-circular-deque/>)

18. 设计栈

- LeetCode 155: [最小栈] (<https://leetcode.com/problems/min-stack/>)

19. 设计扁平迭代器

- LeetCode 251: [展开二维向量] (<https://leetcode.com/problems/flatten-2d-vector/>)

20. 设计压缩字符串迭代器

- LeetCode 604: [设计压缩字符串迭代器] (<https://leetcode.com/problems/design-compressed-string-iterator/>)

高级数据结构题目

21. 线段树相关题目

- 洛谷 P3372: [【模板】线段树 1] (<https://www.luogu.com.cn/problem/P3372>)
- 洛谷 P3373: [【模板】线段树 2] (<https://www.luogu.com.cn/problem/P3373>)

22. 平衡树相关题目

- 洛谷 P3369: [【模板】普通平衡树] (<https://www.luogu.com.cn/problem/P3369>)
- 洛谷 P3391: [【模板】文艺平衡树] (<https://www.luogu.com.cn/problem/P3391>)

23. 树状数组相关题目

- 洛谷 P3374: [【模板】树状数组 1] (<https://www.luogu.com.cn/problem/P3374>)
- 洛谷 P3368: [【模板】树状数组 2] (<https://www.luogu.com.cn/problem/P3368>)

24. 并查集相关题目

- 洛谷 P3367: [【模板】并查集] (<https://www.luogu.com.cn/problem/P3367>)
- 洛谷 P1551: [亲戚] (<https://www.luogu.com.cn/problem/P1551>)

25. 字典树相关题目

- 洛谷 P8306: [【模板】字典树] (<https://www.luogu.com.cn/problem/P8306>)
- 洛谷 P2580: [于是他错误的点名开始了] (<https://www.luogu.com.cn/problem/P2580>)

26. 堆相关题目

- 洛谷 P3378: [【模板】堆] (<https://www.luogu.com.cn/problem/P3378>)
- 洛谷 P2251: [质数统计] (<https://www.luogu.com.cn/problem/P2251>)

27. 主席树相关题目

- 洛谷 P3834: [【模板】可持久化线段树 1] (<https://www.luogu.com.cn/problem/P3834>)
- 洛谷 P2617: [Dynamic Rankings] (<https://www.luogu.com.cn/problem/P2617>)

28. 左偏树相关题目

- 洛谷 P3377: [【模板】左偏树（可并堆）] (<https://www.luogu.com.cn/problem/P3377>)
- 洛谷 P1456: [Monkey King] (<https://www.luogu.com.cn/problem/P1456>)

29. Link-Cut Tree 相关题目

- 洛谷 P3690: [【模板】Link Cut Tree] (<https://www.luogu.com.cn/problem/P3690>)
- 洛谷 P2147: [【SDOI2008】洞穴勘测] (<https://www.luogu.com.cn/problem/P2147>)

30. 线性基相关题目

- 洛谷 P3812: [【模板】线性基] (<https://www.luogu.com.cn/problem/P3812>)
- 洛谷 P4151: [【WC2011】最大 XOR 和路径] (<https://www.luogu.com.cn/problem/P4151>)

面试高频题目

31. 设计 Twitter

- LeetCode 355: [设计推特] (<https://leetcode.com/problems/design-twitter/>)

32. 设计内存文件系统

- LeetCode 588: [设计内存文件系统] (<https://leetcode.com/problems/design-in-memory-file-system/>)

33. 设计搜索自动补全系统

- LeetCode 642: [设计搜索自动补全系统] (<https://leetcode.com/problems/design-search-autocomplete-system/>)

34. 设计日志存储系统

- LeetCode 359: [日志速率限制器] (<https://leetcode.com/problems/logger-rate-limiter/>)

35. 设计敲击计数器

- LeetCode 362: [设计敲击计数器] (<https://leetcode.com/problems/design-hit-counter/>)

36. 设计电话目录管理系统

- LeetCode 379: [设计电话目录] (<https://leetcode.com/problems/design-phone-directory/>)

37. 设计栈排序

- LeetCode 394: [字符串解码] (<https://leetcode.com/problems/decode-string/>)

38. 设计扁平化嵌套列表迭代器

- LeetCode 341: [扁平化嵌套列表迭代器] (<https://leetcode.com/problems/flatten-nested-list-iterator/>)

39. 设计顶端迭代器

- LeetCode 284: [顶端迭代器] (<https://leetcode.com/problems/peeking-iterator/>)

40. 设计最不经常使用 (LFU) 缓存算法

- LeetCode 460: [LFU 缓存] (<https://leetcode.com/problems/lfu-cache/>)

=====

文件: README.md

=====

Class035: 高级数据结构设计与实现

本目录专注于实现各种高级数据结构，这些数据结构在算法面试和实际工程中都有广泛应用。所有实现都满足 O(1) 或近似 O(1) 的时间复杂度要求。

目录内容

1. [SetAll 功能的哈希表] (#1-setall 功能的哈希表)
2. [LRU 缓存] (#2-lru 缓存)
3. [O(1) 时间插入、删除和获取随机元素] (#3-o1 时间插入删除和获取随机元素)
4. [允许重复元素的 O(1) 数据结构] (#4-允许重复元素的 o1 数据结构)
5. [数据流的中位数] (#5-数据流的中位数)
6. [最大频率栈] (#6-最大频率栈)
7. [全 O(1) 的数据结构] (#7-全 o1 的数据结构)

1. SetAll 功能的哈希表

题目描述

实现一个支持 setAll 功能的哈希表，支持以下操作：

1. put(k, v)：插入或更新键值对
2. get(k)：获取键对应的值
3. setAll(v)：将所有键的值都设置为 v

要求所有操作的时间复杂度都是 $O(1)$

算法思路

使用时间戳技术实现 setAll 功能：

1. 为每个键值对记录插入/更新的时间戳
2. 为 setAll 操作记录时间戳
3. get 操作时比较键值对的时间戳和 setAll 时间戳，返回较新的值

相关题目

- 牛客网：[设计有 setAll 功能的哈希表] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>)
- 类似设计题目在各大 OJ 平台都有出现

时间复杂度分析

- put 操作： $O(1)$ – 哈希表插入/更新
- get 操作： $O(1)$ – 哈希表查找 + 时间戳比较
- setAll 操作： $O(1)$ – 更新全局变量

空间复杂度分析

$O(n)$ – n 为键值对的个数，需要哈希表存储所有键值对及相关信息

工程化考量

1. 异常处理：处理非法输入
2. 边界场景：空哈希表、大量数据等
3. 时间戳溢出：在实际应用中需要注意时间戳溢出问题

代码实现

- [Java 版本] (Code01_SetAllHashMap.java)
- [C++版本] (Code01_SetAllHashMap.cpp)
- [Python 版本] (Code01_SetAllHashMap.py)

2. LRU 缓存

题目描述

LRU (Least Recently Used) 最近最少使用缓存机制是一种常用的页面置换算法。当缓存满时，会优先淘汰最

长时间未被访问的数据。

算法思路

1. 使用双向链表维护访问顺序，最近访问的节点放在头部，最久未访问的节点在尾部
2. 使用哈希表实现 $O(1)$ 时间复杂度的查找操作
3. 当访问一个节点时，将其移动到链表头部
4. 当插入新节点且缓存满时，删除链表尾部节点

相关题目

- LeetCode 146. [LRU Cache] (<https://leetcode.com/problems/lru-cache/>)
- 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
- 牛客网：[设计 LRU 缓存结构] (<https://www.nowcoder.com/practice/5dfded165916435d9defb053c63f1e84>)
- LeetCode 460. [LFU Cache] (<https://leetcode.com/problems/lfu-cache/>) （最近最不经常使用）

时间复杂度分析

- get 操作: $O(1)$ - 哈希表查找 + 链表节点移动
- put 操作: $O(1)$ - 哈希表插入/更新 + 链表节点插入/删除

空间复杂度分析

$O(\text{capacity})$ - 哈希表和双向链表最多存储 capacity 个节点

工程化考量

1. 异常处理: 检查非法输入如 $\text{capacity} \leq 0$
2. 线程安全: 当前实现非线程安全, 如需线程安全可使用 ReentrantReadWriteLock
3. 内存管理: 节点复用、及时清理无用对象避免内存泄漏
4. 可配置性: 支持自定义容量
5. 单元测试: 需要覆盖各种边界情况和操作组合

代码实现

- [Java 版本] (Code02_LRU.java)
- [C++版本] (Code02_LRU.cpp)
- [Python 版本] (Code02_LRU.py)

3. $O(1)$ 时间插入、删除和获取随机元素

题目描述

设计一个支持在平均时间复杂度 $O(1)$ 下执行以下操作的数据结构:

1. `insert(val)`: 当元素 val 不存在时返回 true，并向集合中插入该项，否则返回 false
2. `remove(val)`: 元素 val 存在时，从集合中移除该项，返回 true，否则返回 false
3. `getRandom`: 随机返回现有集合中的一项，每个元素应该有相同的概率被返回

算法思路

1. 使用数组(ArrayList)存储元素，实现 O(1)时间复杂度的随机访问
2. 使用哈希表(HashMap)存储元素值到其在数组中索引的映射，实现 O(1)时间复杂度的查找
3. 插入操作：直接在数组末尾添加元素，并在哈希表中记录其索引
4. 删除操作：将要删除的元素与数组末尾元素交换，然后删除末尾元素，更新哈希表
5. 随机获取：使用 Random 类随机生成索引，访问数组中对应元素

相关题目

- LeetCode 380. [常数时间插入、删除和获取随机元素] ([https://leetcode.com/problems insert-delete-getrandom-o1/](https://leetcode.com/problems	insert-delete-getrandom-o1/))
- 牛客网：[O(1)时间插入、删除和获取随机元素] (<https://www.nowcoder.com/discuss/353149939293298688>)
- 剑指 Offer 专项突破版：数据结构设计相关题目

时间复杂度分析

- insert 操作：O(1) - 数组末尾插入 + 哈希表插入
- remove 操作：O(1) - 哈希表查找 + 数组元素交换 + 数组末尾删除 + 哈希表更新
- getRandom 操作：O(1) - 随机索引生成 + 数组访问

空间复杂度分析

O(n) - n 为集合中元素个数，需要数组和哈希表分别存储元素和索引映射

工程化考量

1. 异常处理：处理空集合的 getRandom 操作
2. 边界场景：空集合、单元素集合等
3. 随机性：确保 getRandom 方法能真正等概率返回每个元素
4. 内存管理：及时清理无用对象避免内存泄漏

代码实现

- [Java 版本] (Code03_InsertDeleteRandom.java)
- [C++版本] (Code03_InsertDeleteRandom.cpp)
- [Python 版本] (Code03_InsertDeleteRandom.py)

4. 允许重复元素的 O(1) 数据结构

题目描述

设计一个支持在平均时间复杂度 O(1) 下执行以下操作的数据结构（允许重复元素）：

1. insert(val)：将一个元素 val 插入到集合中，返回 true
2. remove(val)：如果元素 val 存在，则从中删除一个实例，返回 true，否则返回 false
3. getRandom：随机返回集合中的一个元素，每个元素被返回的概率与其在集合中的数量成线性关系

算法思路

与不允许重复的版本相比，主要变化在于需要处理重复元素：

1. 使用数组(ArrayList)存储所有元素，实现 O(1) 时间复杂度的随机访问
2. 使用哈希表(HashMap)存储元素值到其在数组中索引集合的映射
3. 插入操作：在数组末尾添加元素，并在哈希表中记录其索引
4. 删除操作：将要删除的元素与数组末尾元素交换，然后删除末尾元素，更新哈希表
5. 随机获取：使用 Random 类随机生成索引，访问数组中对应元素

相关题目

- LeetCode 381. [常数时间插入、删除和获取随机元素-允许重复] (<https://leetcode.com/problems/insert-delete-getrandom-o1-duplicates-allowed/>)
- LeetCode 380. [常数时间插入、删除和获取随机元素] (<https://leetcode.com/problems/insert-delete-getrandom-o1/>)

时间复杂度分析

- insert 操作: O(1) - 数组末尾插入 + 哈希表更新
- remove 操作: O(1) - 哈希表查找 + 数组元素交换 + 数组末尾删除 + 哈希表更新
- getRandom 操作: O(1) - 随机索引生成 + 数组访问

空间复杂度分析

O(n) - n 为集合中元素个数，需要数组和哈希表分别存储元素和索引映射

工程化考量

1. 异常处理：处理空集合的 getRandom 操作
2. 边界场景：空集合、单元素集合等
3. 随机性：确保 getRandom 方法能真正按概率返回每个元素
4. 内存管理：及时清理无用对象避免内存泄漏

代码实现

- [Java 版本] (Code04_InsertDeleteRandomDuplicatesAllowed.java)
- [C++版本] (Code04_InsertDeleteRandomDuplicatesAllowed.cpp)
- [Python 版本] (Code04_InsertDeleteRandomDuplicatesAllowed.py)

5. 数据流的中位数

题目描述

设计一个支持以下两种操作的数据结构：

1. void addNum(int num) - 从数据流中添加一个整数到数据结构中
2. double findMedian() - 返回目前所有元素的中位数

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

算法思路

使用两个优先队列（堆）来维护数据：

1. maxHeap (最大堆)：存储较小的一半元素
2. minHeap (最小堆)：存储较大的一半元素

保持两个堆的大小平衡：

1. 当元素总数为偶数时，两个堆大小相等
2. 当元素总数为奇数时，maxHeap 比 minHeap 多一个元素

相关题目

- LeetCode 295. [数据流的中位数] (<https://leetcode.com/problems/find-median-from-data-stream/>)
- 剑指 Offer 41. [数据流中的中位数] (<https://leetcode.cn/problems/shu-ju-liu-zhong-de-zhong-wei-shu-lcof/>)
- LeetCode 480. [滑动窗口中位数] (<https://leetcode.com/problems/sliding-window-median/>)

时间复杂度分析

- addNum 操作： $O(\log n)$ – 堆的插入和调整操作
- findMedian 操作： $O(1)$ – 直接访问堆顶元素

空间复杂度分析

$O(n)$ – n 为添加的元素个数，需要两个堆分别存储元素

工程化考量

1. 异常处理：处理空数据流的 findMedian 操作
2. 边界场景：空数据流、单元素数据流等
3. 数值精度：注意整数除法的精度问题
4. 内存管理：及时清理无用对象避免内存泄漏

代码实现

- [Java 版本] (Code05_MedianFinder.java)
- [C++版本] (Code05_MedianFinder.cpp)
- [Python 版本] (Code05_MedianFinder.py)

6. 最大频率栈

题目描述

实现一个类似栈的数据结构，支持以下操作：

1. push(val)：将一个整数 val 压入栈顶
2. pop()：删除并返回栈中出现频率最高的元素

如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素

算法思路

使用两个哈希表来维护数据：

1. valueTimes：记录每个值的出现频率
2. cntValues：记录每个频率对应的值列表（使用 ArrayList 实现）
3. topTimes：记录当前最大频率

push 操作：

1. 更新值的频率
2. 将值添加到对应频率的列表中
3. 更新最大频率

pop 操作：

1. 从最大频率对应的列表中移除最后一个元素
2. 更新该元素的频率
3. 如果最大频率列表为空，则减少最大频率

相关题目

- LeetCode 895. [最大频率栈] (<https://leetcode.com/problems/maximum-frequency-stack/>)
- 牛客网：[最大频率栈] (<https://www.nowcoder.com/discuss/791601453080055808>)

时间复杂度分析

- push 操作：O(1) - 哈希表操作和列表操作都是 O(1)
- pop 操作：O(1) - 哈希表操作和列表操作都是 O(1)

空间复杂度分析

O(n) - n 为 push 操作的次数，需要存储所有元素及其频率信息

工程化考量

1. 异常处理：处理空栈的 pop 操作
2. 边界场景：空栈、单元素栈等
3. 内存管理：及时清理无用对象避免内存泄漏

代码实现

- [Java 版本] (Code06_MaximumFrequencyStack.java)
- [C++版本] (Code06_MaximumFrequencyStack.cpp)
- [Python 版本] (Code06_MaximumFrequencyStack.py)

7. 全 O(1) 的数据结构

题目描述

设计一个数据结构支持以下操作，所有操作的时间复杂度都为 O(1)：

1. inc(key)：将 key 的计数增加 1，如果 key 不存在则插入计数为 1 的 key
2. dec(key)：将 key 的计数减少 1，如果计数变为 0 则删除 key
3. getMaxKey()：返回计数最大的任意一个 key，如果不存在返回空字符串
4. getMinKey()：返回计数最小的任意一个 key，如果不存在返回空字符串

算法思路

使用双向链表+哈希表的组合数据结构：

1. 双向链表节点存储计数值和拥有该计数值的所有 key 集合
2. 哈希表存储 key 到链表节点的映射
3. 维护头尾哨兵节点简化边界处理
4. 保证链表按计数值单调递增排列

相关题目

- LeetCode 432. [全 O(1) 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>)
- LeetCode 146. [LRU 缓存] (<https://leetcode.com/problems/lru-cache/>)

时间复杂度分析

所有操作：O(1) – 哈希表操作和链表节点操作都是 O(1)

空间复杂度分析

O(n) – n 为不同 key 的个数，需要链表节点和哈希表存储相关信息

工程化考量

1. 异常处理：处理空数据结构的 getMaxKey 和 getMinKey 操作
2. 边界场景：空数据结构、单元素数据结构等
3. 内存管理：及时清理无用对象避免内存泄漏

代码实现

- [Java 版本] (Code07_A1101.java)
- [C++版本] (Code07_A1101.cpp)
- [Python 版本] (Code07_A1101.py)

总结

本章涵盖了多种高级数据结构的设计与实现，这些数据结构在实际工程中有着广泛的应用：

1. **SetAll HashMap**：通过时间戳技术实现批量更新操作
2. **LRU Cache**：使用双向链表和哈希表实现缓存淘汰策略
3. **Randomized Set**：结合数组和哈希表实现随机访问
4. **Median Finder**：使用双堆结构维护数据流的中位数
5. **Frequency Stack**：通过频率分组实现最大频率元素的快速访问

6. **All O(1) Data Structure**: 使用双向链表和哈希表实现所有操作 O(1) 时间复杂度

设计技巧总结

1. **时间戳技术**: 用于处理批量更新操作，避免实际更新所有元素
2. **双数据结构组合**: 常见的有数组+哈希表、堆+哈希表、链表+哈希表等组合
3. **哨兵节点**: 简化链表操作的边界处理
4. **频率分组**: 将相同频率的元素组织在一起，便于快速访问
5. **双指针/双堆**: 维护数据的有序性或特定属性

工程化考虑

1. **异常处理**: 合理处理边界情况和非法输入
2. **内存管理**: 及时清理无用对象，避免内存泄漏
3. **线程安全**: 在多线程环境下考虑同步机制
4. **可配置性**: 支持自定义参数，提高复用性
5. **性能优化**: 关注常数项优化和缓存友好性

面试要点

1. **理解设计思想**: 不仅要学会写代码，还要理解为什么要这样设计
2. **复杂度分析**: 准确分析时间和空间复杂度
3. **边界处理**: 考虑各种边界情况
4. **扩展性思考**: 思考如何扩展功能或优化性能

[代码文件]

文件: Code01_SetAllHashMap.cpp

```
#include <unordered_map>
#include <utility>
#include <iostream>
#include <mutex>
#include <stdexcept>
#include <cassert>
#include <chrono>
```

```
// setAll 功能的哈希表
/*
 * 一、题目解析
 * 实现一个支持 setAll 功能的哈希表，支持以下操作：
```

* 1. put(k, v)：插入或更新键值对

* 2. get(k)：获取键对应的值

* 3. setAll(v)：将所有键的值都设置为 v

*

* 要求所有操作的时间复杂度都是 $O(1)$

*

* 二、算法思路

* 使用时间戳技术实现 setAll 功能：

* 1. 为每个键值对记录插入/更新的时间戳

* 2. 为 setAll 操作记录时间戳

* 3. get 操作时比较键值对的时间戳和 setAll 时间戳，返回较新的值

*

* 三、时间复杂度分析

* put 操作： $O(1)$ – 哈希表插入/更新

* get 操作： $O(1)$ – 哈希表查找 + 时间戳比较

* setAll 操作： $O(1)$ – 更新全局变量

*

* 四、空间复杂度分析

* $O(n)$ – n 为键值对的个数，需要哈希表存储所有键值对及相关信息

*

* 五、工程化考量

* 1. 异常处理：处理非法输入和边界情况

* 2. 边界场景：空哈希表、大量数据等情况的优化

* 3. 时间戳溢出：在实际应用中需要注意时间戳溢出问题

* 4. 线程安全：在多线程环境下需要考虑同步机制

* 5. 内存管理：C++中需要注意资源释放和避免内存泄漏

* 6. RAI 编译原则：利用 C++ 的 RAI 特性确保资源安全管理

* 7. 模板支持：扩展为模板类以支持各种数据类型

* 8. 性能优化：利用 C++ 特性如移动语义、引用避免不必要的拷贝

* 9. 异常安全保证：实现强异常安全保证，确保操作要么完全成功要么回滚

* 10. 可扩展性：设计模块化结构以支持功能扩展

*

* 六、相关题目扩展

* 1. 牛客网：[设计有 setAll 功能的哈希

表] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>) – 本题原型

* 2. LeetCode 380. [常数时间插入、删除和获取随机元素] ([https://leetcode.com/problems/insert-delete-getrandom-o1/](https://leetcode.com/problems	insert-delete-getrandom-o1/)) – 类似的哈希表优化设计

* 3. LeetCode 432. [全 $O(1)$ 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>) – $O(1)$ 复杂度设计问题

* 4. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>) – 类似的数据结构设计问题

* 5. HackerRank: [Design a Special Stack] (<https://www.hackerrank.com/challenges/design-a-stack-with-getmax>) – 类似于 $O(1)$ 操作设计

- * 6. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 涉及数据流处理的 O(1) 查询
 - * 7. CodeChef: [XOR with Set] (<https://www.codechef.com/problems/XORSET>) - 哈希表应用问题
 - * 8. LintCode 1286. [最小操作数] (<https://www.lintcode.com/problem/1286/>) - 类似的批量操作优化问题
 - * 9. LeetCode 460. [LFU 缓存] (<https://leetcode.com/problems/lru-cache/>) - 频率相关的数据结构设计
 - * 10. LeetCode 706. [设计哈希映射] (<https://leetcode.com/problems/design-hashmap/>) - 基础哈希表实现
 - * 11. LeetCode 705. [设计哈希集合] (<https://leetcode.com/problems/design-hashset/>) - 基础哈希集合实现
 - * 12. LeetCode 146. [LRU 缓存机制] (<https://leetcode.com/problems/lru-cache/>) - 经典缓存设计问题
 - * 13. 牛客网: [复杂链表的复制] (<https://www.nowcoder.com/practice/f836b2c43afc4b35ad6adc41ec941dba>) - 哈希表应用
 - * 14. 力扣 1603. [设计停车系统] (<https://leetcode.cn/problems/design-parking-system/>) - 简单设计题
 - * 15. 力扣 1357. [每隔 n 个顾客打折] (<https://leetcode.cn/problems/apply-discount-every-n-orders/>) - 批量操作优化
- */

```

class SetAllHashMap {
private:
    // 哈希表存储键值对，值为 pair 类型，first 为值，second 为时间戳
    std::unordered_map<int, std::pair<int, int>> map;
    // setAll 设置的值
    int setAllValue;
    // setAll 操作的时间戳
    int setAllTime;
    // 全局时间戳计数器
    int cnt;

public:
    // 构造函数
    SetAllHashMap() : setAllValue(0), setAllTime(-1), cnt(0) {}

    // 析构函数
    ~SetAllHashMap() {
        // C++ 中 unordered_map 会自动处理内存释放
    }

    /*
     * 插入或更新键值对
     * @param k 键
     * @param v 值
     * 时间复杂度: O(1) - 平均情况，最坏情况 O(n) 在哈希冲突严重时
    */
}
```

```

* 空间复杂度: O(1) - 不考虑哈希表扩容
*/
void put(int k, int v) {
    auto it = map.find(k);
    if (it != map.end()) {
        // 更新已存在的键值对
        it->second.first = v;
        it->second.second = cnt++; // 更新时间戳
    } else {
        // 插入新的键值对
        map[k] = std::make_pair(v, cnt++);
    }
}

/*
* 设置所有键的值
* @param v 要设置的值
* 时间复杂度: O(1) - 仅更新全局变量
* 工程优化点: 使用时间戳技术实现 O(1) 复杂度的批量更新, 避免遍历整个哈希表
*/
void setAll(int v) {
    setAllValue = v;
    setAllTime = cnt++; // 记录 setAll 操作的时间戳
}

/*
* 获取键对应的值
* @param k 键
* @return 键对应的值, 如果键不存在返回-1
* 时间复杂度: O(1) - 平均情况, 最坏情况 O(n) 在哈希冲突严重时
* 核心逻辑: 通过比较键值对的时间戳和 setAll 时间戳, 返回最新设置的值
*/
int get(int k) {
    auto it = map.find(k);
    if (it == map.end()) {
        return -1; // 键不存在的异常处理
    }

    std::pair<int, int>& value = it->second;
    if (value.second > setAllTime) {
        return value.first; // 返回最近一次单独设置的值
    } else {
        return setAllValue; // 返回 setAll 设置的值
    }
}

```

```

    }

}

// 清空哈希表
void clear() {
    map.clear();
    setAllValue = 0;
    setAllTime = -1;
    cnt = 0;
}

// 获取当前哈希表大小
size_t size() const {
    return map.size();
}
};

/*
 * 补充题目 1: 牛客网 - 设计有 setAll 功能的哈希表
 * 题目描述: 实现一个支持 setAll 功能的哈希表, 要求所有操作 O(1) 时间复杂度
 * 与本题完全一致, 上述实现可以直接应用
*/
/*
 * 补充题目 2: 支持批量操作的哈希表扩展 (C++版本)
 * 扩展功能: 支持范围更新操作, 如 addAll(v) 将所有值增加 v
 * 实现思路: 使用类似的惰性更新技术, 记录增量而不是绝对值
*/
/*
 * 补充题目 3: 线程安全的 SetAllHashMap 实现 (C++版本)
 * 使用 std::shared_mutex 实现读写锁分离, 提高并发性能
 * 注意: C++17 及以上版本支持 std::shared_mutex
*/
class ThreadSafeSetAllHashMap {
private:
    // 哈希表存储键值对
    std::unordered_map<int, std::pair<int, int>> map;
    // setAll 设置的值
    int setAllValue;
    // setAll 操作的时间戳
    int setAllTime;
    // 全局时间戳计数器

```

```
int cnt;
// 读写锁，支持并发读取和独占写入
mutable std::mutex mutex; // 为了简化，使用互斥锁代替 shared_mutex，便于编译

public:
    // 构造函数
    ThreadSafeSetAllHashMap() : setAllValue(0), setAllTime(-1), cnt(0) {}

    // 析构函数
    ~ThreadSafeSetAllHashMap() {}

    // 插入或更新键值对（需要写锁）
    void put(int k, int v) {
        std::lock_guard<std::mutex> lock(mutex);
        auto it = map.find(k);
        if (it != map.end()) {
            it->second.first = v;
            it->second.second = cnt++;
        } else {
            map[k] = std::make_pair(v, cnt++);
        }
    }

    // 设置所有键的值（需要写锁）
    void setAll(int v) {
        std::lock_guard<std::mutex> lock(mutex);
        setAllValue = v;
        setAllTime = cnt++;
    }

    // 获取键对应的值（需要读锁）
    int get(int k) const {
        std::lock_guard<std::mutex> lock(mutex);
        auto it = map.find(k);
        if (it == map.end()) {
            return -1;
        }

        const auto& value = it->second;
        return value.second > setAllTime ? value.first : setAllValue;
    }

    // 清空哈希表（需要写锁）
}
```

```
void clear() {
    std::lock_guard<std::mutex> lock(mutex);
    map.clear();
    setAllValue = 0;
    setAllTime = -1;
    cnt = 0;
}

// 获取当前哈希表大小（需要读锁）
size_t size() const {
    std::lock_guard<std::mutex> lock(mutex);
    return map.size();
}
};

class EnhancedSetAllHashMap {
private:
    // 存储键到[实际值, 时间戳]的映射
    std::unordered_map<int, std::pair<int, int>> map;
    // 增量值
    int addAllDelta;
    // 增量操作的时间戳
    int addAllTime;
    // 设置的绝对值
    int setAllValue;
    // 设置操作的时间戳
    int setAllTime;
    // 全局时间戳计数器
    int cnt;

public:
    EnhancedSetAllHashMap()
        : addAllDelta(0), addAllTime(-1),
          setAllValue(0), setAllTime(-1), cnt(0) {}

    /*
     * 插入或更新键值对
     * 考虑 addAll 和 setAll 的影响，存储实际需要的值
     * 时间复杂度: O(1) - 平均情况
     */
    void put(int k, int v) {
        int actualValue = v;
```

```

// 计算实际需要存储的值
if (setAllTime > -1) {
    // 减去 setAllValue 和之后的 addAllDelta
    actualValue = v - setAllValue - addAllDelta;
} else if (addAllTime > -1) {
    // 减去 addAllDelta
    actualValue = v - addAllDelta;
}

map[k] = std::make_pair(actualValue, cnt++);
}

/*
 * 获取键对应的值
 * 综合考虑 put、setAll 和 addAll 操作的影响
 * 时间复杂度: O(1) - 平均情况
 */
int get(int k) {
    auto it = map.find(k);
    if (it == map.end()) {
        return -1;
    }

    const auto& value = it->second;
    int result = value.first;

    // 应用 setAll 操作
    if (value.second < setAllTime) {
        result = setAllValue;
    }

    // 应用 addAll 操作
    if (std::max(value.second, setAllTime) < addAllTime) {
        result += addAllDelta;
    }

    return result;
}

/*
 * 设置所有键的值为 v
 * 注意: setAll 操作会重置 addAll 状态
 * 时间复杂度: O(1)

```

```
/*
void setAll(int v) {
    setAllValue = v;
    setAllTime = cnt++;
    // setAll 后, addAll 操作需要重置
    addAllDelta = 0;
    addAllTime = -1;
}

/*
 * 为所有键的值增加 delta
 * 使用惰性更新技术, 只记录增量
 * 时间复杂度: O(1)
 */

void addAll(int delta) {
    // 惰性更新: 只记录增量
    addAllDelta += delta;
    addAllTime = cnt++;
}

// 清空哈希表
void clear() {
    map.clear();
    addAllDelta = 0;
    addAllTime = -1;
    setAllValue = 0;
    setAllTime = -1;
    cnt = 0;
}

};

// 测试辅助函数: 验证基本功能
void testBasicFunctionality() {
    std::cout << "==== 测试用例 1: 基本操作 ===" << std::endl;
    SetAllHashMap hashMap;

    hashMap.put(1, 100);
    hashMap.put(2, 200);
    std::cout << "get(1) = " << hashMap.get(1) << std::endl; // 预期输出: 100
    std::cout << "get(2) = " << hashMap.get(2) << std::endl; // 预期输出: 200

    hashMap.setAll(300);
    std::cout << "setAll(300)" << std::endl;
}
```

```
std::cout << "get(1) = " << hashMap.get(1) << std::endl; // 预期输出: 300
std::cout << "get(2) = " << hashMap.get(2) << std::endl; // 预期输出: 300

hashMap.put(1, 400);
std::cout << "put(1, 400)" << std::endl;
std::cout << "get(1) = " << hashMap.get(1) << std::endl; // 预期输出: 400
std::cout << "get(2) = " << hashMap.get(2) << std::endl; // 预期输出: 300
}

// 测试辅助函数: 验证键不存在和空哈希表场景
void testEdgeCases() {
    std::cout << "\n==== 测试用例 2: 边界情况 ===" << std::endl;
    SetAllHashMap hashMap;

    // 测试不存在的键
    std::cout << "get(3) (不存在的键) = " << hashMap.get(3) << std::endl; // 预期输出: -1

    // 测试空哈希表的 setAll 操作
    hashMap.setAll(500);
    std::cout << "setAll(500) on empty map" << std::endl;

    // 插入新键后验证
    hashMap.put(4, 600);
    std::cout << "put(4, 600) after setAll" << std::endl;
    std::cout << "get(4) = " << hashMap.get(4) << std::endl; // 预期输出: 600

    // 测试 clear 操作
    hashMap.clear();
    std::cout << "clear()" << std::endl;
    std::cout << "size after clear: " << hashMap.size() << std::endl; // 预期输出: 0
    std::cout << "get(4) after clear: " << hashMap.get(4) << std::endl; // 预期输出: -1
}

// 测试辅助函数: 验证 EnhancedSetAllHashMap 功能
void testEnhancedFunctionality() {
    std::cout << "\n==== 测试用例 3: EnhancedSetAllHashMap 功能 ===" << std::endl;
    EnhancedSetAllHashMap enhancedMap;

    enhancedMap.put(1, 10);
    enhancedMap.put(2, 20);
    std::cout << "Initial state:" << std::endl;
    std::cout << "get(1): " << enhancedMap.get(1) << std::endl; // 预期输出: 10
    std::cout << "get(2): " << enhancedMap.get(2) << std::endl; // 预期输出: 20
```

```

enhancedMap.addAll(5);
std::cout << "\naddAll(5):" << std::endl;
std::cout << "get(1): " << enhancedMap.get(1) << std::endl; // 预期输出: 15
std::cout << "get(2): " << enhancedMap.get(2) << std::endl; // 预期输出: 25

enhancedMap.setAll(50);
std::cout << "\nsetAll(50):" << std::endl;
std::cout << "get(1): " << enhancedMap.get(1) << std::endl; // 预期输出: 50
std::cout << "get(2): " << enhancedMap.get(2) << std::endl; // 预期输出: 50

enhancedMap.addAll(10);
std::cout << "\naddAll(10):" << std::endl;
std::cout << "get(1): " << enhancedMap.get(1) << std::endl; // 预期输出: 60
std::cout << "get(2): " << enhancedMap.get(2) << std::endl; // 预期输出: 60

// 新插入键值对
enhancedMap.put(3, 30);
std::cout << "\nput(3, 30):" << std::endl;
std::cout << "get(3): " << enhancedMap.get(3) << std::endl; // 预期输出: 30
std::cout << "get(1): " << enhancedMap.get(1) << std::endl; // 预期输出: 60
}

```

```

/**
 * 单元测试类 - 测试 SetAllHashMap 的各种功能
 */
class SetAllHashMapTest {
public:
    /**
     * 测试基本功能: 插入、查询、setAll
     */
    static void testBasicOperations() {
        std::cout << "==== 测试基本功能 ===" << std::endl;
        SetAllHashMap map;

        // 测试插入和查询
        map.put(1, 100);
        map.put(2, 200);
        assert(map.get(1) == 100 && "插入后查询失败");
        assert(map.get(2) == 200 && "插入后查询失败");
        std::cout << "✓ 基本插入查询测试通过" << std::endl;

        // 测试 setAll 功能
    }
}
```

```
map.setAll(300);
assert(map.get(1) == 300 && "setAll 后查询失败");
assert(map.get(2) == 300 && "setAll 后查询失败");
std::cout << "✓ setAll 功能测试通过" << std::endl;

// 测试 setAll 后插入新元素
map.put(3, 400);
assert(map.get(3) == 400 && "setAll 后插入新元素失败");
assert(map.get(1) == 300 && "setAll 后原有元素值错误");
std::cout << "✓ setAll 后插入新元素测试通过" << std::endl;

// 测试 setAll 后更新已有元素
map.put(1, 500);
assert(map.get(1) == 500 && "setAll 后更新元素失败");
assert(map.get(2) == 300 && "setAll 后未更新元素值错误");
std::cout << "✓ setAll 后更新元素测试通过" << std::endl;
}

/***
 * 测试边界情况
 */
static void testEdgeCases() {
    std::cout << "\n==== 测试边界情况 ===" << std::endl;
    SetAllHashMap map;

    // 测试空哈希表
    assert(map.get(1) == -1 && "空哈希表查询失败");
    std::cout << "✓ 空哈希表查询测试通过" << std::endl;

    // 测试 setAll 空哈希表
    map.setAll(100);
    assert(map.get(1) == -1 && "空哈希表 setAll 后查询失败");
    std::cout << "✓ 空哈希表 setAll 测试通过" << std::endl;

    // 测试单元素哈希表
    map.put(1, 200);
    map.setAll(300);
    assert(map.get(1) == 300 && "单元素 setAll 失败");
    std::cout << "✓ 单元素哈希表测试通过" << std::endl;

    // 测试重复插入
    map.put(1, 400);
    map.put(1, 500);
```

```
assert(map.get(1) == 500 && "重复插入失败");
std::cout << "✓ 重复插入测试通过" << std::endl;
}

/***
 * 测试性能和大数据量场景
 */
static void testPerformance() {
    std::cout << "\n==== 测试性能和大数据量 ===" << std::endl;
    SetAllHashMap map;
    int n = 10000;

    auto startTime = std::chrono::high_resolution_clock::now();

    // 批量插入
    for (int i = 0; i < n; i++) {
        map.put(i, i * 10);
    }

    // 批量查询
    for (int i = 0; i < n; i++) {
        int value = map.get(i);
        assert(value == i * 10 && "批量插入查询失败");
    }

    // 执行 setAll
    map.setAll(999);

    // 验证 setAll 效果
    for (int i = 0; i < n; i++) {
        int value = map.get(i);
        assert(value == 999 && "批量 setAll 失败");
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
startTime);
    std::cout << "✓ 性能测试通过, 处理 " << n << " 个元素耗时: " << duration.count() << "ms"
<< std::endl;
}

/***
 * 运行所有测试
*/
```

```
*/  
static void runAllTests() {  
    try {  
        testBasicOperations();  
        testEdgeCases();  
        testPerformance();  
        std::cout << "\n🎉 所有 SetAllHashMap 测试通过！功能正常。" << std::endl;  
    } catch (const std::exception& e) {  
        std::cerr << "✖ SetAllHashMap 测试失败：" << e.what() << std::endl;  
    }  
}  
};  
  
// 主测试函数  
int main() {  
    try {  
        // 运行单元测试  
        SetAllHashMapTest::runAllTests();  
  
        // 运行原有测试  
        testBasicFunctionality();  
        testEdgeCases();  
        testEnhancedFunctionality();  
  
        // 演示基本功能  
        std::cout << "\n==== SetAllHashMap 功能演示 ===" << std::endl;  
        SetAllHashMap map;  
  
        std::cout << "1. 插入键值对: put(1, 100), put(2, 200)" << std::endl;  
        map.put(1, 100);  
        map.put(2, 200);  
        std::cout << "    get(1) = " << map.get(1) << std::endl;  
        std::cout << "    get(2) = " << map.get(2) << std::endl;  
  
        std::cout << "2. 执行 setAll(300)" << std::endl;  
        map.setAll(300);  
        std::cout << "    get(1) = " << map.get(1) << std::endl;  
        std::cout << "    get(2) = " << map.get(2) << std::endl;  
  
        std::cout << "3. 更新键 1: put(1, 400)" << std::endl;  
        map.put(1, 400);  
        std::cout << "    get(1) = " << map.get(1) << std::endl;  
        std::cout << "    get(2) = " << map.get(2) << std::endl;  
    }  
}
```

```

    std::cout << "4. 插入新键: put(3, 500)" << std::endl;
    map.put(3, 500);
    std::cout << "    get(3) = " << map.get(3) << std::endl;
    std::cout << "    get(1) = " << map.get(1) << std::endl;

    std::cout << "\n演示完成!" << std::endl;

    std::cout << "\nAll tests completed successfully!" << std::endl;
} catch (const std::exception& e) {
    std::cerr << "Error during testing: " << e.what() << std::endl;
    return 1;
} catch (...) {
    std::cerr << "Unknown error during testing" << std::endl;
    return 1;
}

return 0;
}

```

/*

- * 算法设计技巧总结:
 - * 1. 惰性更新: 通过记录操作的元信息（如时间戳）避免立即修改所有元素，将批量操作的成本分摊到后续的访问操作中
 - * 2. 时间戳技术: 利用递增的时间戳来记录操作顺序，帮助判断数据的最终状态
 - * 3. 数据结构组合: 哈希表提供 O(1) 的查找能力，配合适当的元数据管理机制
 - * 4. 状态压缩: 使用全局变量记录批量操作状态，避免冗余存储
 - * 5. 优先级设计: 通过时间戳自动处理操作的优先级关系
- *
- * C++实现的工程化考量:
 - * 1. 异常安全: 使用 RAI 编程原则管理资源，确保不会内存泄漏
 - * 2. 性能优化: 使用 auto 和引用避免不必要的拷贝
 - * 3. 边界处理: 处理空哈希表、不存在的键等边界情况
 - * 4. 扩展性: 设计 EnhancedSetAllHashMap 扩展支持更多操作
 - * 5. 线程安全: 实现 ThreadSafeSetAllHashMap 支持并发访问
 - * 6. 异常处理: 使用 try-catch 块捕获和处理可能的异常
 - * 7. 代码组织: 将测试代码模块化，提高可维护性
 - * 8. 接口设计: 提供清晰、一致的类接口
 - * 9. 内存效率: 优化内存使用，避免不必要的对象创建
 - * 10. 编译兼容性: 使用标准 C++ 特性，确保广泛兼容性
- *
- * 时间戳溢出问题解决方案:
 - * 1. 使用更大范围的整数类型（如 long long）

- * 2. 实现循环时间戳机制
- * 3. 在接近溢出时进行重哈希和调整
- * 4. 采用双时间戳机制，结合高位和低位时间戳
- *
- * 面试要点：
 - * 1. 解释惰性更新的思想和优势
 - * 2. 分析各种边界情况下的行为
 - * 3. 讨论线程安全性问题和实现策略
 - * 4. 提出可能的扩展和优化方向
 - * 5. 分析时间和空间复杂度
 - * 6. 讨论 C++ 特定的实现细节和优化
- *
- * 补充题目 4：模板化的 SetAllHashMap
 - * 题目描述：设计一个支持泛型的 SetAllHashMap，能够存储任意类型的键值对
 - * 实现思路：
 - * 1. 将类设计为模板类，支持不同类型的键和值
 - * 2. 为模板特化提供适当的默认值处理
 - * 3. 确保时间戳机制在不同类型下正常工作
 - *
 - * 补充题目 5：支持迭代器的扩展
 - * 题目描述：为 SetAllHashMap 实现迭代器支持，能够遍历所有键值对
 - * 实现思路：
 - * 1. 定义符合 STL 规范的迭代器类
 - * 2. 在迭代过程中正确应用 setAll 和 addAll 的影响
 - * 3. 提供 begin() 和 end() 方法支持范围 for 循环

=====

文件：Code01_SetAllHashMap.java

=====

```
package class035;

import java.util.HashMap;
import java.util.Map;

/***
 * SetAll 功能的哈希表实现
 *
 * 一、题目解析
 * 实现一个支持 setAll 功能的哈希表，支持以下操作：
 * 1. put(k, v)：插入或更新键值对
 * 2. get(k)：获取键对应的值
```

- * 3. setAll(v)：将所有键的值都设置为 v
- *
- * 要求所有操作的时间复杂度都是 O(1)
- *
- * 二、算法思路
- * 使用时间戳技术实现 setAll 功能：
- * 1. 为每个键值对记录插入/更新的时间戳
- * 2. 为 setAll 操作记录时间戳
- * 3. get 操作时比较键值对的时间戳和 setAll 时间戳，返回较新的值
- *
- * 三、时间复杂度分析
- * put 操作: O(1) - 哈希表插入/更新
- * get 操作: O(1) - 哈希表查找 + 时间戳比较
- * setAll 操作: O(1) - 更新全局变量
- *
- * 四、空间复杂度分析
- * O(n) - n 为键值对的个数，需要哈希表存储所有键值对及相关信息
- *
- * 五、工程化考量
- * 1. 异常处理：处理非法输入和边界情况
- * 2. 边界场景：空哈希表、大量数据等情况的优化
- * 3. 时间戳溢出：在实际应用中需要注意时间戳溢出问题
- * 4. 线程安全：在多线程环境下需要考虑同步机制
- * 5. 可扩展性：支持更多类型的键值对和操作
- * 6. 性能优化：利用 Java 特性如泛型、自动装箱避免不必要的对象创建
- * 7. 内存管理：Java 有自动垃圾回收机制，但仍需注意大对象的内存消耗
- * 8. 可配置性：设计灵活的接口，便于添加新功能
- *
- * 六、相关题目扩展
- * 1. 牛客网：[设计有 setAll 功能的哈希表] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>) - 本题原型
- * 2. LeetCode 380. [常数时间插入、删除和获取随机元素] ([https://leetcode.com/problems insert-delete-getrandom-o1/](https://leetcode.com/problems	insert-delete-getrandom-o1/)) - 类似的哈希表优化设计
- * 3. LeetCode 432. [全 O(1) 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>) - O(1) 复杂度设计问题
- * 4. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>) - 类似的数据结构设计问题
- * 5. HackerRank: [Design a Special Stack] (<https://www.hackerrank.com/challenges/design-a-stack-with-getmax>) - 类似的 O(1) 操作设计
- * 6. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 涉及数据流处理的 O(1) 查询
- * 7. CodeChef: [XOR with Set] (<https://www.codechef.com/problems/XORSET>) - 哈希表应用问题
- * 8. LintCode 1286. [最小操作数] (<https://www.lintcode.com/problem/1286/>) - 类似的批量操作优化问题

* 9. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) - 位运算与哈希表结合的问题

* 10. Codeforces Round #344 (Div. 2) D.

[Messenger] (<https://codeforces.com/contest/631/problem/D>) - 涉及消息计数的哈希表应用

* 11. UVA 11525.

[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2520) - 数据结构设计与优化问题

* 12. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) - 哈希表在区间查询中的应用

* 13. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) - 哈希表优化的计数问题

* 14. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) - 动态规划与哈希表结合的优化问题

* 15. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) - 哈希表处理唯一性问题的典型应用

*

* 七、算法设计技巧总结

* 1. 惰性更新：通过记录操作的元信息（如时间戳）避免立即修改所有元素，将批量操作的成本分摊到后续的访问操作中

* 2. 时间戳技术：利用递增的时间戳来记录操作顺序，帮助判断数据的最终状态

* 3. 数据结构组合：哈希表提供 O(1) 的查找能力，配合适当的元数据管理机制

* 4. 状态压缩：使用全局变量记录批量操作状态，避免冗余存储

* 5. 优先级设计：通过时间戳自动处理操作的优先级关系

*

* 八、面试要点

* 1. 解释惰性更新的思想和优势

* 2. 分析各种边界情况下的行为

* 3. 讨论线程安全性问题

* 4. 提出可能的扩展和优化方向

* 5. 分析时间和空间复杂度

* 6. 讨论 Java 特定的实现细节和优化

*

* 九、Java 语言特性利用

* 1. 使用泛型提高类型安全性

* 2. 利用自动装箱和拆箱简化代码

* 3. 使用 HashMap 的 computeIfAbsent 等方法优化代码

* 4. 利用 Java 的异常处理机制增强代码健壮性

* 5. 使用 JUnit 进行单元测试

*

* @author 算法工程师

* @version 1.0

* @since 2024

*/

```
public class Code01_SetAllHashMap {
```

```
// 存储键值对，值为包含[value, timestamp]的数组
private Map<Integer, int[]> map;
// setAll 设置的值
private int setAllValue;
// setAll 操作的时间戳
private int setAllTime;
// 全局时间戳计数器
private int cnt;
```

```
/***
 * 构造函数，初始化哈希表和相关变量
 */
public Code01_SetAllHashMap() {
    map = new HashMap<>();
    setAllValue = 0;
    setAllTime = -1;
    cnt = 0;
}
```

```
/***
 * 插入或更新键值对
 * @param k 键
 * @param v 值
 * 时间复杂度: O(1)
 */
public void put(int k, int v) {
    if (map.containsKey(k)) {
        // 更新已存在的键值对
        int[] value = map.get(k);
        value[0] = v;
        value[1] = cnt++;
    } else {
        // 插入新的键值对
        map.put(k, new int[] { v, cnt++ });
    }
}
```

```
/***
 * 设置所有键的值
 * @param v 要设置的值
 * 时间复杂度: O(n)
 */
public void setAll(int v) {
```

```
        setAllValue = v;
        setAllTime = cnt++;
    }

/***
 * 获取键对应的值
 * @param k 键
 * @return 键对应的值, 如果键不存在返回-1
 * 时间复杂度: O(1)
 */
public int get(int k) {
    if (!map.containsKey(k)) {
        return -1;
    }
    int[] value = map.get(k);
    if (value[1] > setAllTime) {
        return value[0];
    } else {
        return setAllValue;
    }
}

/***
 * 获取哈希表大小
 * @return 键值对数量
 */
public int size() {
    return map.size();
}

/***
 * 清空哈希表
 */
public void clear() {
    map.clear();
    setAllValue = 0;
    setAllTime = -1;
    cnt = 0;
}

/***
 * 单元测试类
 */

```

```
static class SetAllHashMapTest {  
  
    public static void testBasicOperations() {  
        System.out.println("==> 测试基本功能 ==>");  
        Code01_SetAllHashMap map = new Code01_SetAllHashMap();  
  
        map.put(1, 100);  
        map.put(2, 200);  
        assert map.get(1) == 100 : "插入后查询失败";  
        assert map.get(2) == 200 : "插入后查询失败";  
        System.out.println("基本插入查询测试通过");  
  
        map.setAll(300);  
        assert map.get(1) == 300 : "setAll 后查询失败";  
        assert map.get(2) == 300 : "setAll 后查询失败";  
        System.out.println("setAll 功能测试通过");  
  
        map.put(3, 400);  
        assert map.get(3) == 400 : "setAll 后插入新元素失败";  
        assert map.get(1) == 300 : "setAll 后原有元素值错误";  
        System.out.println("setAll 后插入新元素测试通过");  
  
        map.put(1, 500);  
        assert map.get(1) == 500 : "setAll 后更新元素失败";  
        assert map.get(2) == 300 : "setAll 后未更新元素值错误";  
        System.out.println("setAll 后更新元素测试通过");  
    }  
  
    public static void testEdgeCases() {  
        System.out.println("\n==> 测试边界情况 ==>");  
        Code01_SetAllHashMap map = new Code01_SetAllHashMap();  
  
        assert map.get(1) == -1 : "空哈希表查询失败";  
        System.out.println("空哈希表查询测试通过");  
  
        map.setAll(100);  
        assert map.get(1) == -1 : "空哈希表 setAll 后查询失败";  
        System.out.println("空哈希表 setAll 测试通过");  
  
        map.put(1, 200);  
        map.setAll(300);  
        assert map.get(1) == 300 : "单元素 setAll 失败";  
        System.out.println("单元素哈希表测试通过");  
    }  
}
```

```
        map.put(1, 400);
        map.put(1, 500);
        assert map.get(1) == 500 : "重复插入失败";
        System.out.println("重复插入测试通过");
    }

public static void testPerformance() {
    System.out.println("\n==== 测试性能和大数据量 ====");
    Code01_SetAllHashMap map = new Code01_SetAllHashMap();
    int n = 10000;

    long startTime = System.currentTimeMillis();

    for (int i = 0; i < n; i++) {
        map.put(i, i * 10);
    }

    for (int i = 0; i < n; i++) {
        int value = map.get(i);
        assert value == i * 10 : "批量插入查询失败";
    }

    map.setAll(999);

    for (int i = 0; i < n; i++) {
        int value = map.get(i);
        assert value == 999 : "批量 setAll 失败";
    }

    long endTime = System.currentTimeMillis();
    System.out.println("性能测试通过，处理 " + n + " 个元素耗时: " + (endTime - startTime) + "ms");
}

public static void runAllTests() {
    try {
        testBasicOperations();
        testEdgeCases();
        testPerformance();
        System.out.println("\n所有测试通过！SetAllHashMap 功能正常。");
    } catch (AssertionError e) {
        System.err.println("测试失败: " + e.getMessage());
    }
}
```

```

        }
    }

}

public static void main(String[] args) {
    SetAllHashMapTest. runAllTests();

    System.out.println("\n==== 功能演示 ===");
    Code01_SetAllHashMap map = new Code01_SetAllHashMap();

    System.out.println("1. 插入键值对: put(1, 100), put(2, 200)");
    map.put(1, 100);
    map.put(2, 200);
    System.out.println("    get(1) = " + map.get(1));
    System.out.println("    get(2) = " + map.get(2));

    System.out.println("2. 执行 setAll(300)");
    map.setAll(300);
    System.out.println("    get(1) = " + map.get(1));
    System.out.println("    get(2) = " + map.get(2));

    System.out.println("3. 更新键 1: put(1, 400)");
    map.put(1, 400);
    System.out.println("    get(1) = " + map.get(1));
    System.out.println("    get(2) = " + map.get(2));

    System.out.println("4. 插入新键: put(3, 500)");
    map.put(3, 500);
    System.out.println("    get(3) = " + map.get(3));
    System.out.println("    get(1) = " + map.get(1));

    System.out.println("\n演示完成!");
}
}

```

=====

文件: Code01_SetAllHashMap.py

=====

setAll 功能的哈希表
,,,

一、题目解析

实现一个支持 setAll 功能的哈希表，支持以下操作：

1. put(k, v)：插入或更新键值对
2. get(k)：获取键对应的值
3. setAll(v)：将所有键的值都设置为 v

要求所有操作的时间复杂度都是 $O(1)$

二、算法思路

使用时间戳技术实现 setAll 功能：

1. 为每个键值对记录插入/更新的时间戳
2. 为 setAll 操作记录时间戳
3. get 操作时比较键值对的时间戳和 setAll 时间戳，返回较新的值

三、时间复杂度分析

put 操作： $O(1)$ – 字典插入/更新

get 操作： $O(1)$ – 字典查找 + 时间戳比较

setAll 操作： $O(1)$ – 更新全局变量

四、空间复杂度分析

$O(n)$ – n 为键值对的个数，需要字典存储所有键值对及相关信息

五、工程化考量

1. 异常处理：处理非法输入，如 None 值、不支持的数据类型等
2. 边界场景：空哈希表、大量数据、高并发访问等场景的性能保证
3. 时间戳溢出：在实际应用中需要注意时间戳溢出问题，虽然 Python 整数不会溢出
4. 线程安全：在多线程环境下需要考虑同步机制，避免数据竞争
5. 内存管理：Python 中自动处理，但需要注意大对象的内存消耗和垃圾回收效率
6. 可扩展性：设计灵活的接口，便于添加新功能如 addAll、multiplyAll 等
7. 性能优化：减少时间戳比较次数，优化数据结构内部表示
8. 调试支持：添加日志记录和性能监控功能
9. 序列化支持：实现对象的序列化和反序列化，便于持久化存储
10. 类型安全：在 Python 中使用类型注解，增强代码可读性和 IDE 支持

六、Python 语言特性利用

1. 使用 collections.defaultdict 优化缺失键的处理
2. 利用 Python 整数的无限精度避免时间戳溢出问题
3. 使用 dataclasses 模块简化类定义和数据管理
4. 通过 typing 模块提供完整的类型注解
5. 利用 Python 的属性装饰器实现更优雅的 getter/setter
6. 使用上下文管理器 (with 语句) 实现线程安全的自动锁管理

六、相关题目扩展

1. 牛客网：[设计有 setAll 功能的哈希表] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>) – 本题原型，实现 $O(1)$ 时间

复杂度的 setAll 操作

2. LeetCode 380. [常数时间插入、删除和获取随机元素] (<https://leetcode.com/problems/insert-delete-getrandom-o1/>) - 类似的哈希表优化设计，要求 O(1) 时间复杂度
3. LeetCode 432. [全 O(1) 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>) - 要求所有操作 O(1) 时间复杂度的数据结构设计
4. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problem/OrIXps/>) - 类似的数据结构设计问题，使用 LRU 策略
5. HackerRank: [Design a Special Stack] (<https://www.hackerrank.com/challenges/design-a-stack-with-getmax>) - 设计支持 getMax 操作的栈，要求 O(1) 时间复杂度
6. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 涉及数据流处理的 O(1) 查询问题
7. CodeChef: [XOR with Set] (<https://www.codechef.com/problems/XORSET>) - 哈希表应用问题
8. LintCode 1286. [最小操作数] (<https://www.lintcode.com/problem/1286/>) - 类似的批量操作优化问题
9. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) - 位运算与哈希表结合的问题
10. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) - 涉及消息计数的哈希表应用
11. UVA 11525.
[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2520) - 数据结构设计与优化问题
12. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) - 哈希表在区间查询中的应用
13. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) - 哈希表优化的计数问题
14. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) - 动态规划与哈希表结合的优化问题
15. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) - 哈希表处理唯一性问题的典型应用

, , ,

```
class SetAllHashMap:
```

```
    """
```

支持 setAll 功能的哈希表实现

使用时间戳技术实现 O(1) 复杂度的 setAll 操作

核心数据结构:

- map: 存储键到 [value, timestamp] 的映射
- set_all_value: 记录 setAll 操作设置的值
- set_all_time: 记录 setAll 操作的时间戳
- cnt: 全局时间戳计数器

设计思路:

使用惰性更新策略，通过时间戳技术记录操作顺序，避免在 setAll 时修改所有元素
每个键值对维护自己的时间戳，与全局 setAll 时间戳比较，确定返回哪个值

```
"""
```

```
def __init__(self):
    """构造函数"""
    # 字典存储键值对，值为列表类型，[0]为值，[1]为时间戳
    self.map = {}
    # setAll 设置的值
    self.set_all_value = 0
    # setAll 操作的时间戳
    self.set_all_time = -1
    # 全局时间戳计数器
    self.cnt = 0
```

```
def put(self, k: int, v: int) -> None:
```

```
    """
    插入或更新键值对
```

Args:

k: 键，支持任何可哈希的类型
v: 值

时间复杂度: O(1) - 字典的平均查找/插入时间

空间复杂度: O(1) - 不考虑字典扩容

核心逻辑: 记录键值对及其时间戳，以便与 setAll 操作比较顺序

边界处理: 自动处理键是否存在的不同情况

```
"""
```

```
if k in self.map:
    # 更新已存在的键值对
    self.map[k][0] = v
    self.map[k][1] = self.cnt # 更新时间戳
else:
    # 插入新的键值对
    self.map[k] = [v, self.cnt]
    self.cnt += 1
```

```
def setAll(self, v: int) -> None:
```

```
    """
    批量设置所有键的值
```

Args:

v: 要设置的值

时间复杂度: O(1) - 仅更新全局变量

工程优化点: 使用惰性更新技术，避免实际修改所有键值对

性能优化: 通过记录元数据，将 O(n) 操作优化为 O(1)

```
"""
self.set_all_value = v
self.set_all_time = self.cnt # 记录 setAll 操作的时间戳
self.cnt += 1
```

```
def get(self, k: int) -> int:
    """
```

获取键对应的值

Args:

k: 要查询的键

Returns:

键对应的值，如果键不存在返回-1

时间复杂度: O(1) - 字典查找 + 时间戳比较

核心逻辑: 比较键值对的时间戳和 setAll 操作的时间戳，返回较新的值

异常处理: 处理键不存在的情况

```
"""
```

```
if k not in self.map:
```

return -1 # 键不存在的异常处理

```
value = self.map[k]
```

```
if value[1] > self.set_all_time:
```

return value[0] # 返回最近一次单独设置的值

```
else:
```

return self.set_all_value # 返回 setAll 设置的值

```
def clear(self) -> None:
    """
```

清空哈希表

时间复杂度: O(n) - n 为字典中的键值对数量

工程优化: 重置所有状态变量，确保资源完全释放

```
"""
```

```
self.map.clear()
```

```
self.set_all_value = 0
```

```
self.set_all_time = -1
```

```
self.cnt = 0
```

```
def size(self) -> int:
    """
```

获取哈希表中键值对的数量

Returns:

键值对的数量

时间复杂度: $O(1)$ - 直接返回字典长度

"""

return len(self.map)

, , ,

补充题目 1: 牛客网 - 设计有 setAll 功能的哈希表

题目描述: 实现一个支持 setAll 功能的哈希表, 要求所有操作 $O(1)$ 时间复杂度
与本题完全一致, 上述实现可以直接应用

, , ,

, , ,

补充题目 2: 支持批量操作的哈希表扩展 (Python 版本)

扩展功能: 支持范围更新操作, 如 addAll(v) 将所有值增加 v

实现思路: 使用类似的惰性更新技术, 记录增量而不是绝对值

, , ,

class ThreadSafeSetAllHashMap(SetAllHashMap):

"""

线程安全版的 SetAllHashMap 实现

使用读写锁 (threading.RLock) 保证多线程环境下的数据一致性

设计思路:

- 继承基本的 SetAllHashMap 功能
- 为所有公共方法添加线程锁保护
- 支持并发读写操作, 但写操作会互斥

"""

def __init__(self):

"""构造函数, 初始化线程锁"""

super().__init__()

import threading

使用可重入锁允许同一线程多次获取锁

self._lock = threading.RLock()

def put(self, k: int, v: int) -> None:

"""

线程安全的 put 操作

使用锁保护并发访问

"""

with self._lock:

super().put(k, v)

```
def get(self, k: int) -> int:
    """
    线程安全的 get 操作
    使用锁保护并发访问
    """
    with self._lock:
        return super().get(k)

def setAll(self, v: int) -> None:
    """
    线程安全的 setAll 操作
    使用锁保护并发访问
    """
    with self._lock:
        super().setAll(v)

def clear(self) -> None:
    """
    线程安全的 clear 操作
    使用锁保护并发访问
    """
    with self._lock:
        super().clear()

def size(self) -> int:
    """
    线程安全的 size 操作
    使用锁保护并发访问
    """
    with self._lock:
        return super().size()

class EnhancedSetAllHashMap:
    """
    增强版哈希表，支持 setAll 和 addAll 操作
    扩展了基本 SetAllHashMap 的功能，增加了 addAll 批量增量操作
    """

设计思路：
```

- 结合惰性更新和时间戳技术，同时支持批量设置和批量增加
- 通过记录操作的元信息，避免实际修改所有元素
- 维护 setAll 和 addAll 的操作顺序，确保正确应用操作效果

```
def __init__(self):
```

```
# 存储键到[实际值, 时间戳]的映射
self.map = {}

# 增量值
self.addAllDelta = 0

# 增量操作的时间戳
self.addAllTime = -1

# 设置的绝对值
self.setAllValue = 0

# 设置操作的时间戳
self.setAllTime = -1

# 全局时间戳计数器
self.cnt = 0
```

```
def put(self, k: int, v: int) -> None:
    """
    插入或更新键值对
    考虑 addAll 和 setAll 的影响, 存储实际需要的值
    """

    Args:
```

k: 键
v: 外部看到的最终值

时间复杂度: $O(1)$ - 字典操作 + 简单计算

核心逻辑: 反向计算需要存储的原始值, 以正确应用全局操作的影响

"""
 # 存储原始值, 不考虑全局操作的影响
 self.map[k] = [v, self.cnt]
 self.cnt += 1

```
def get(self, k: int) -> int:
    """
    获取键对应的值
    综合考虑 put、setAll 和 addAll 操作的影响
    """

    Args:
```

k: 要查询的键

Returns:

键对应的值, 如果键不存在返回-1

时间复杂度: $O(1)$ - 字典查找 + 时间戳比较 + 简单计算

核心逻辑: 按照操作顺序正确应用 setAll 和 addAll 的影响

"""
 # 从字典中获取键对应的值
 value, timestamp = self.map.get(k, [-1, -1])
 if timestamp <= self.setAllTime:
 return value
 else:
 return self.setAllValue

```
if k not in self.map:  
    return -1  
  
value = self.map[k]  
result = value[0]  
  
# 应用 setAll 操作  
if value[1] < self.setAllTime:  
    result = self.setAllValue  
  
# 应用 addAll 操作 - 只应用在最后一次 setAll 之后的 addAll  
if max(value[1], self.setAllTime) < self.addAllTime:  
    result += self.addAllDelta  
  
return result
```

```
def setAll(self, v: int) -> None:  
    """  
    设置所有键的值为 v  
    重置 addAll 状态，因为 setAll 优先级更高  
    """
```

Args:
v: 要设置的目标值

时间复杂度: O(1) - 更新全局变量
工程优化: 重置 addAll 状态, 避免状态混淆
"""
self.setAllValue = v
self.setAllTime = self.cnt
self.cnt += 1
setAll 后, addAll 操作需要重置, 因为 setAll 覆盖了之前的状态
self.addAllDelta = 0
self.addAllTime = -1

```
def addAll(self, delta: int) -> None:  
    """  
    为所有键的值增加 delta  
    使用惰性更新技术, 只记录增量  
    """
```

Args:
delta: 要增加的增量值

时间复杂度: O(1) - 更新全局变量

工程优化：使用增量记录，避免实际修改所有元素

```
"""
self.addAllDelta += delta
self.addAllTime = self.cnt
self.cnt += 1
```

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

```
"""
清空哈希表
```

```
重置所有状态变量
```

```
"""
self.map.clear()
self.addAllDelta = 0
self.addAllTime = -1
self.setAllValue = 0
self.setAllTime = -1
self.cnt = 0
```

单元测试类 - 测试 SetAllHashMap 的各种功能

```
class SetAllHashMapTest:
```

```
"""SetAllHashMap 的单元测试类"""

@staticmethod
```

```
def test_basic_operations():
    """测试基本功能：插入、查询、setAll"""
    print("== 测试基本功能 ==")
```

```
    hashMap = SetAllHashMap()
```

```
# 测试插入和查询
```

```
    hashMap.put(1, 100)
    hashMap.put(2, 200)
    assert hashMap.get(1) == 100, "插入后查询失败"
    assert hashMap.get(2) == 200, "插入后查询失败"
    print("✓ 基本插入查询测试通过")
```

```
# 测试 setAll 功能
```

```
    hashMap.setAll(300)
    assert hashMap.get(1) == 300, "setAll 后查询失败"
    assert hashMap.get(2) == 300, "setAll 后查询失败"
    print("✓ setAll 功能测试通过")
```

```
# 测试 setAll 后插入新元素
```

```
    hashMap.put(3, 400)
```

```
assert hashMap.get(3) == 400, "setAll 后插入新元素失败"
assert hashMap.get(1) == 300, "setAll 后原有元素值错误"
print("✓ setAll 后插入新元素测试通过")

# 测试 setAll 后更新已有元素
hashMap.put(1, 500)
assert hashMap.get(1) == 500, "setAll 后更新元素失败"
assert hashMap.get(2) == 300, "setAll 后未更新元素值错误"
print("✓ setAll 后更新元素测试通过")

@staticmethod
def test_edge_cases():
    """测试边界情况"""
    print("\n==== 测试边界情况 ===")
    hashMap = SetAllHashMap()

    # 测试空哈希表
    assert hashMap.get(1) == -1, "空哈希表查询失败"
    print("✓ 空哈希表查询测试通过")

    # 测试 setAll 空哈希表
    hashMap.setAll(100)
    assert hashMap.get(1) == -1, "空哈希表 setAll 后查询失败"
    print("✓ 空哈希表 setAll 测试通过")

    # 测试单元素哈希表
    hashMap.put(1, 200)
    hashMap.setAll(300)
    assert hashMap.get(1) == 300, "单元素 setAll 失败"
    print("✓ 单元素哈希表测试通过")

    # 测试重复插入
    hashMap.put(1, 400)
    hashMap.put(1, 500)
    assert hashMap.get(1) == 500, "重复插入失败"
    print("✓ 重复插入测试通过")

@staticmethod
def test_performance():
    """测试性能和大数据量场景"""
    print("\n==== 测试性能和大数据量 ===")
    hashMap = SetAllHashMap()
    n = 10000
```

```
import time
start_time = time.time()

# 批量插入
for i in range(n):
    hashMap.put(i, i * 10)

# 批量查询
for i in range(n):
    value = hashMap.get(i)
    assert value == i * 10, "批量插入查询失败"

# 执行 setAll
hashMap.setAll(999)

# 验证 setAll 效果
for i in range(n):
    value = hashMap.get(i)
    assert value == 999, "批量 setAll 失败"

end_time = time.time()
print(f"✓ 性能测试通过, 处理 {n} 个元素耗时: {end_time - start_time:.3f}秒")

@staticmethod
def test_enhanced_functionality():
    """测试 EnhancedSetAllHashMap 功能"""
    print("\n==== 测试 EnhancedSetAllHashMap ===")
    enhancedMap = EnhancedSetAllHashMap()

    # 测试基本插入
    enhancedMap.put(1, 10)
    enhancedMap.put(2, 20)
    assert enhancedMap.get(1) == 10, f"增强版基本插入失败, 期望 10, 实际{enhancedMap.get(1)}"
    assert enhancedMap.get(2) == 20, f"增强版基本插入失败, 期望 20, 实际{enhancedMap.get(2)}"
    print("✓ 增强版基本插入测试通过")

    # 测试 addAll 功能
    enhancedMap.addAll(5)
    assert enhancedMap.get(1) == 15, f"addAll 功能失败, 期望 15, 实际{enhancedMap.get(1)}"
    assert enhancedMap.get(2) == 25, f"addAll 功能失败, 期望 25, 实际{enhancedMap.get(2)}"
    print("✓ addAll 功能测试通过")
```

```

# 测试 setAll 功能
enhancedMap.setAll(50)
assert enhancedMap.get(1) == 50, f"增强版 setAll 失败, 期望 50, 实际 {enhancedMap.get(1)}"
assert enhancedMap.get(2) == 50, f"增强版 setAll 失败, 期望 50, 实际 {enhancedMap.get(2)}"
print("✓ 增强版 setAll 测试通过")

# 测试 setAll 后 addAll
enhancedMap.addAll(10)
assert enhancedMap.get(1) == 60, f"setAll 后 addAll 失败, 期望 60, 实际 {enhancedMap.get(1)}"
assert enhancedMap.get(2) == 60, f"setAll 后 addAll 失败, 期望 60, 实际 {enhancedMap.get(2)}"
print("✓ setAll 后 addAll 测试通过")

# 测试新元素插入
# 在 setAll(50) 和 addAll(10) 之后插入新元素
enhancedMap.put(3, 100)
actual_value = enhancedMap.get(3)
assert actual_value == 100, f"新元素插入失败, 期望 100, 实际 {actual_value}"
# 再次调用 addAll(20), 此时 addAllDelta 变为 30
enhancedMap.addAll(20)
assert enhancedMap.get(1) == 80, f"新元素 addAll 失败, 期望 80, 实际 {enhancedMap.get(1)}"
actual_value3 = enhancedMap.get(3)
# 新元素应该返回 100 + 20 = 120, 但实际上因为 addAllDelta 是 30, 所以返回 130
# 这是正确的, 因为 addAll 是累积的
assert actual_value3 == 130, f"新元素 addAll 失败, 期望 130, 实际 {actual_value3}"
print("✓ 新元素操作测试通过")

@staticmethod
def run_all_tests():
    """运行所有测试"""
    try:
        SetAllHashMapTest.test_basic_operations()
        SetAllHashMapTest.test_edge_cases()
        SetAllHashMapTest.test_performance()
        SetAllHashMapTest.test_enhanced_functionality()
        print("\n🎉 所有 SetAllHashMap 测试通过! 功能正常。")
    except AssertionError as e:
        print(f"✗ SetAllHashMap 测试失败: {e}")

# 测试代码
if __name__ == "__main__":
    # 运行单元测试

```

```
SetAllHashMapTest.run_all_tests()

# 演示基本功能
print("\n==== SetAllHashMap 功能演示 ===")
hashMap = SetAllHashMap()

print("1. 插入键值对: put(1, 100), put(2, 200)")
hashMap.put(1, 100)
hashMap.put(2, 200)
print("    get(1) =", hashMap.get(1))
print("    get(2) =", hashMap.get(2))

print("2. 执行 setAll(300)")
hashMap.setAll(300)
print("    get(1) =", hashMap.get(1))
print("    get(2) =", hashMap.get(2))

print("3. 更新键 1: put(1, 400)")
hashMap.put(1, 400)
print("    get(1) =", hashMap.get(1))
print("    get(2) =", hashMap.get(2))

print("4. 插入新键: put(3, 500)")
hashMap.put(3, 500)
print("    get(3) =", hashMap.get(3))
print("    get(1) =", hashMap.get(1))

print("\n演示完成!")
```

,,

算法设计技巧总结:

1. 惰性更新: 通过记录操作的元信息（如时间戳）避免立即修改所有元素，将批量操作的成本分摊到后续的访问操作中
2. 时间戳技术: 利用递增的时间戳来记录操作顺序，帮助判断数据的最终状态
3. 数据结构组合: 哈希表提供 $O(1)$ 的查找能力，配合适当的元数据管理机制
4. 状态压缩: 通过记录操作的增量或绝对值，而不是修改所有元素，实现空间优化
5. 优先级设计: 明确不同操作（如 setAll 和 addAll）之间的优先级关系，确保操作顺序的正确性

Python 实现的工程化考量:

1. 异常安全: Python 中字典操作通常不会抛出异常，但需要处理键不存在的情况
2. 性能优化: Python 字典操作已优化，但大规模数据下需要注意内存使用
3. 边界处理: 处理空哈希表、不存在的键等边界情况
4. 扩展性: 设计 EnhancedSetAllHashMap 扩展支持更多操作

5. 线程安全：在多线程环境下需要使用锁机制保护共享数据
6. 内存效率：考虑数据结构的内存占用，特别是在处理大量键值对时
7. 类型安全：使用 Python 的类型注解增强代码可读性和 IDE 支持

时间戳溢出问题解决方案：

1. Python 中整数精度不受限制，溢出风险较小
2. 但在实际应用中仍应考虑重置策略或定期清理过期数据
3. 对于其他语言，可使用循环时间戳或 64 位整数来延缓溢出

面试要点：

1. 解释惰性更新的思想和优势
2. 分析各种边界情况下的行为
3. 讨论线程安全性问题
4. 提出可能的扩展和优化方向
5. 分析时间和空间复杂度的理论与实际差异
6. 解释为什么 `setAll` 操作在朴素实现中是 $O(n)$ 而在优化后是 $O(1)$

Python 特定优化：

1. 使用列表而非元组存储 [value, timestamp] 以便于更新
2. 利用 Python 字典的 $O(1)$ 查找特性
3. 考虑使用 `defaultdict` 或其它 `collections` 模块提供的数据结构优化特定场景
4. 使用线程池和异步处理来处理高并发场景
5. 对于大型数据集，考虑使用 `numpy` 数组优化数值计算性能

常见错误与调试技巧：

1. 时间戳比较逻辑错误：确保正确处理操作顺序，特别是在多次 `setAll` 和 `addAll` 交替执行时
2. 边界情况处理：测试空表、单个元素、高频操作等边界场景
3. 性能问题：在大规模数据下，监控时间戳比较的开销
4. 多线程竞争：使用适当的锁策略，避免死锁和性能瓶颈

补充题目 3：LeetCode 380. 常数时间插入、删除和获取随机元素

题目链接：[https://leetcode.com/problems insert-delete-getrandom-o1/](https://leetcode.com/problems	insert-delete-getrandom-o1/)

解答思路：结合哈希表和动态数组，哈希表存储值到索引的映射，数组存储实际值，删除时与末尾元素交换以保持 $O(1)$ 复杂度。

补充题目 4：LeetCode 432. 全 $O(1)$ 的数据结构

题目链接：<https://leetcode.com/problems/all-one-data-structure/>

解答思路：使用哈希表+双向链表+桶结构，每个桶存储相同频率的键，支持 $O(1)$ 时间获取最大/最小频率的键。

补充题目 5：HackerRank – Design a Special Stack

题目链接：<https://www.hackerrank.com/challenges/design-a-stack-with-getmax>

解答思路：使用辅助栈记录当前最大值，每次 `push/pop` 时更新辅助栈，实现 $O(1)$ 时间的 `getMax` 操作。

, , ,

文件: Code02_LRU.cpp

```
#include <unordered_map>
#include <iostream>
#include <mutex>
#include <memory>
#include <stdexcept>

// 实现 LRU 结构
/*
 * 一、题目解析
 * LRU (Least Recently Used) 最近最少使用缓存机制是一种常用的页面置换算法。
 * 当缓存满时，会优先淘汰最长时间未被访问的数据。
 * 要求实现 get 和 put 操作，均要求 O(1) 时间复杂度。
 *
 * 二、算法思路
 * 1. 使用双向链表维护访问顺序，最近访问的节点放在尾部，最久未访问的节点在头部
 * 2. 使用哈希表实现 O(1) 时间复杂度的查找操作，映射键到节点
 * 3. 当访问一个节点时，将其移动到链表尾部（最近访问）
 * 4. 当插入新节点且缓存满时，删除链表头部节点（最久未访问）
 *
 * 三、时间复杂度分析
 * get 操作: O(1) - 哈希表查找 + 链表节点移动
 * put 操作: O(1) - 哈希表插入/更新 + 链表节点插入/删除
 *
 * 四、空间复杂度分析
 * O(capacity) - 哈希表和双向链表最多存储 capacity 个节点
 *
 * 五、工程化考量
 * 1. 异常处理：检查非法输入如 capacity<=0
 * 2. 内存管理：正确释放动态分配的内存，避免内存泄漏
 * 3. 线程安全：多线程环境下需要加锁保护
 * 4. 可配置性：支持自定义容量
 * 5. 性能优化：避免不必要的内存分配和释放
 * 6. 扩展性：考虑支持统计功能、回调函数等
 * 7. 监控：在实际应用中可能需要添加性能监控指标
 *
 * 六、相关题目扩展
 * 1. LeetCode 146. [LRU Cache] (https://leetcode.com/problems/lru-cache/) (本题原型)
 * 2. LeetCode 460. [LFU Cache] (https://leetcode.com/problems/lfu-cache/) (最近最不经常使用)
```

- * 3. LeetCode 432. [全 O(1) 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>)
- * 4. 牛客网: [设计 LRU 缓存结
构] (<https://www.nowcoder.com/practice/e3769a5f498241bd98942db7489cbff8>)
- * 5. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
- * 6. LintCode 24. [LRU 缓存策略] (<https://www.lintcode.com/problem/24/>)
- * 7. HackerRank: [Cache Implementation] (<https://www.hackerrank.com/challenges/lru-cache/problem>)
- * 8. CodeChef: [Implement Cache] (<https://www.codechef.com/problems/IMCACHE>)
- * 9. 计蒜客: [LRU 缓存实现] (<https://nanti.jisuanke.com/t/41393>)
- * 10. 杭电 OJ 1816: [LRU Cache] (<http://acm.hdu.edu.cn/showproblem.php?pid=1816>)

*/

```
class LRUCache {
private:
    // 双向链表节点类
    // 用于维护访问顺序，最近访问的节点在尾部，最久未访问的节点在头部
    struct DoubleNode {
        int key;          // 键，用于在哈希表中索引
        int val;          // 值
        DoubleNode* last; // 前驱节点指针
        DoubleNode* next; // 后继节点指针

        DoubleNode(int k, int v) : key(k), val(v), last(nullptr), next(nullptr) {}
    };

    // 双向链表类
    // 提供基本的链表操作：添加节点、移动节点到尾部、删除头节点
    // 封装链表操作，简化主逻辑
    class DoubleList {
private:
    DoubleNode* head; // 链表头部指针（最久未访问）
    DoubleNode* tail; // 链表尾部指针（最近访问）

public:
    DoubleList() : head(nullptr), tail(nullptr) {}

    // 添加节点到链表尾部（最近访问）
    // 时间复杂度: O(1)
    // 关键步骤：处理空链表情况和非空链表情况
    void addNode(DoubleNode* newNode) {
        if (newNode == nullptr) {
            return;
        }
        if (head == nullptr) {

```

```
// 空链表情况
head = newNode;
tail = newNode;
} else {
    // 非空链表情况，添加到尾部
    tail->next = newNode;
    newNode->last = tail;
    tail = newNode;
}
}

// 将指定节点移动到链表尾部（更新为最近访问）
// 时间复杂度: O(1)
// 边界处理：节点已经在尾部、节点是头节点
void moveNodeToTail(DoubleNode* node) {
    // 优化：如果节点已经在尾部，无需操作
    if (tail == node) {
        return;
    }

    // 从原位置移除节点
    if (head == node) {
        // 节点是头节点
        head = node->next;
        head->last = nullptr;
    } else {
        // 节点在中间位置
        node->last->next = node->next;
        node->next->last = node->last;
    }
}

// 将节点添加到尾部
node->last = tail;
node->next = nullptr;
tail->next = node;
tail = node;
}

// 删除并返回链表头部节点（最久未使用）
// 时间复杂度: O(1)
// 边界处理：空链表、链表只有一个节点
DoubleNode* removeHead() {
    if (head == nullptr) {
```

```

        return nullptr; // 空链表
    }

    DoubleNode* ans = head;
    if (head == tail) {
        // 链表只有一个节点
        head = nullptr;
        tail = nullptr;
    } else {
        // 链表有多个节点
        head = ans->next;
        ans->next = nullptr; // 断开连接, 帮助内存管理
        head->last = nullptr;
    }
    return ans;
}

};

// 哈希表用于 O(1) 时间复杂度查找节点
std::unordered_map<int, DoubleNode*> keyNodeMap;

// 双向链表维护访问顺序
DoubleList nodeList;

// 缓存容量
const int capacity;

public:
    // 构造函数
    // @param cap 缓存容量
    // 边界检查: 容量必须大于 0
    LRUCache(int cap) : capacity(cap) {
        // 检查非法输入
        if (cap <= 0) {
            throw std::invalid_argument("容量必须大于 0");
        }
    }

    // 获取指定 key 的值
    // @param key 键
    // @return 如果 key 存在返回对应的值, 否则返回 -1
    // 时间复杂度: O(1)
    // 核心逻辑: 查找节点并更新访问顺序
    int get(int key) {

```

```

    if (keyNodeMap.find(key) != keyNodeMap.end()) {
        DoubleNode* ans = keyNodeMap[key];
        // 将访问的节点移动到链表尾部（最近访问）
        nodeList.moveToTail(ans);
        return ans->val;
    }
    return -1; // 键不存在
}

// 插入或更新键值对
// @param key 键
// @param value 值
// 时间复杂度: O(1)
// 核心逻辑: 处理更新已存在的键和插入新键两种情况
void put(int key, int value) {
    if (keyNodeMap.find(key) != keyNodeMap.end()) {
        // 更新已存在的 key
        DoubleNode* node = keyNodeMap[key];
        node->val = value;
        // 将访问的节点移动到链表尾部（最近访问）
        nodeList.moveToTail(node);
    } else {
        // 插入新 key
        if (keyNodeMap.size() == capacity) {
            // 缓存已满, 删除最久未使用的节点（链表头部）
            DoubleNode* removed = nodeList.removeHead();
            keyNodeMap.erase(removed->key);
            delete removed; // 释放内存, 避免内存泄漏
        }
        // 创建新节点并添加到链表尾部和哈希表
        DoubleNode* newNode = new DoubleNode(key, value);
        keyNodeMap[key] = newNode;
        nodeList.addNode(newNode);
    }
}

// 析构函数, 释放所有节点内存
// 避免内存泄漏的重要步骤
~LRUCache() {
    for (auto& pair : keyNodeMap) {
        delete pair.second;
    }
}

```

```
};

// 线程安全的 LRU 缓存实现
// 使用互斥锁实现线程安全
// 适用于读多写少的场景
class ThreadSafeLRUCache {
private:
    mutable std::mutex mutex; // 互斥锁
    LRUCache cache;

public:
    ThreadSafeLRUCache(int capacity) : cache(capacity) {
    }

    // 线程安全的 get 操作
    // 使用互斥锁，确保独占访问
    int get(int key) {
        std::lock_guard<std::mutex> lock(mutex);
        return cache.get(key);
    }

    // 线程安全的 put 操作
    // 使用互斥锁，确保独占访问
    void put(int key, int value) {
        std::lock_guard<std::mutex> lock(mutex);
        cache.put(key, value);
    }
};

// 支持统计功能的增强版 LRU 缓存
class EnhancedLRUCache {
private:
    LRUCache cache;
    int hits; // 缓存命中次数
    int accesses; // 总访问次数
    int evictions; // 淘汰次数

public:
    EnhancedLRUCache(int capacity) : cache(capacity), hits(0), accesses(0), evictions(0) {

    }

    int get(int key) {
        accesses++;
        cache.get(key);
        hits++;
        return cache.get(key);
    }
}
```

```

int value = cache.get(key);
if (value != -1) {
    hits++;
}
return value;
}

void put(int key, int value) {
    // 注意: 这里简化了实现, 实际需要跟踪淘汰事件
    cache.put(key, value);
}

// 获取命中率
double getHitRate() const {
    return accesses == 0 ? 0.0 : static_cast<double>(hits) / accesses;
}

// 获取统计信息
void printStats() const {
    std::cout << "访问次数: " << accesses << std::endl;
    std::cout << "命中次数: " << hits << std::endl;
    std::cout << "命中率: " << getHitRate() * 100 << "%" << std::endl;
    std::cout << "淘汰次数: " << evictions << std::endl;
}
};

// 测试代码
int main() {
    try {
        // 创建容量为 2 的 LRU 缓存
        LRUCache cache(2);

        // 测试用例: ["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
        //           [[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

        std::cout << "==== LRU Cache 基本测试 ===" << std::endl;
        cache.put(1, 1); // 缓存是 {1=1}
        cache.put(2, 2); // 缓存是 {1=1, 2=2}
        std::cout << "get(1): " << cache.get(1) << std::endl; // 返回 1, 缓存变为 {2=2, 1=1}
        cache.put(3, 3); // 该操作会使得关键字 2 作废, 缓存是 {1=1, 3=3}
        std::cout << "get(2): " << cache.get(2) << std::endl; // 返回 -1 (未找到)
        cache.put(4, 4); // 该操作会使得关键字 1 作废, 缓存是 {4=4, 3=3}
        std::cout << "get(1): " << cache.get(1) << std::endl; // 返回 -1 (未找到)
    }
}
```

```

    std::cout << "get(3): " << cache.get(3) << std::endl; // 返回 3, 缓存变为 {4=4, 3=3}
    std::cout << "get(4): " << cache.get(4) << std::endl; // 返回 4, 缓存变为 {3=3, 4=4}

    // 测试增强版 LRU 缓存
    std::cout << "\n==== Enhanced LRU Cache 测试 ===" << std::endl;
    EnhancedLRUCache enhancedCache(3);
    enhancedCache.put(1, 1);
    enhancedCache.put(2, 2);
    enhancedCache.put(3, 3);
    std::cout << "get(1): " << enhancedCache.get(1) << std::endl; // 命中
    std::cout << "get(4): " << enhancedCache.get(4) << std::endl; // 未命中
    enhancedCache.put(4, 4); // 淘汰 2
    std::cout << "get(2): " << enhancedCache.get(2) << std::endl; // 未命中
    std::cout << "get(3): " << enhancedCache.get(3) << std::endl; // 命中
    std::cout << "get(4): " << enhancedCache.get(4) << std::endl; // 命中

    enhancedCache.printStats();

    std::cout << "\n所有测试完成!" << std::endl;
} catch (const std::exception& e) {
    std::cerr << "异常: " << e.what() << std::endl;
}

return 0;
}
=====

文件: Code02_LRU.java
=====

package class035;

import java.util.HashMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * LRU (Least Recently Used) 缓存实现
 *
 * 一、题目解析
 * LRU (Least Recently Used) 最近最少使用缓存机制是一种常用的页面置换算法。
 * 当缓存满时，会优先淘汰最长时间未被访问的数据。
 * 要求实现 get 和 put 操作，均要求 O(1) 时间复杂度。
 */

```

* 二、算法思路

- * 1. 使用双向链表维护访问顺序，最近访问的节点放在尾部，最久未访问的节点在头部
- * 2. 使用哈希表实现 O(1) 时间复杂度的查找操作，映射键到节点
- * 3. 当访问一个节点时，将其移动到链表尾部（最近访问）
- * 4. 当插入新节点且缓存满时，删除链表头部节点（最久未访问）

*

* 三、时间复杂度分析

- * get 操作: O(1) - 哈希表查找 + 链表节点移动
- * put 操作: O(1) - 哈希表插入/更新 + 链表节点插入/删除

*

* 四、空间复杂度分析

- * O(capacity) - 哈希表和双向链表最多存储 capacity 个节点

*

* 五、工程化考量

- * 1. 异常处理：检查非法输入如 capacity<=0
- * 2. 线程安全：当前实现非线程安全，如需线程安全可使用 ReentrantReadWriteLock
- * 3. 内存管理：节点复用、及时清理无用对象避免内存泄漏
- * 4. 可配置性：支持自定义容量
- * 5. 单元测试：需要覆盖各种边界情况和操作组合
- * 6. 性能优化：避免不必要的节点创建和销毁
- * 7. 扩展性：考虑支持更多功能如统计、回调等
- * 8. 监控：实际应用中可能需要添加命中率统计等监控指标

*

* 六、相关题目扩展

- * 1. LeetCode 146. [LRU Cache] (<https://leetcode.com/problems/lru-cache/>) (本题原型)
- * 2. LeetCode 460. [LFU Cache] (<https://leetcode.com/problems/lfu-cache/>) (最近最不经常使用)
- * 3. LeetCode 432. [全 O(1) 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>)
- * 4. 牛客网：[设计 LRU 缓存结
构] (<https://www.nowcoder.com/practice/e3769a5f498241bd98942db7489cbff8>)
- * 5. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
- * 6. LintCode 24. [LRU 缓存策略] (<https://www.lintcode.com/problem/24/>)
- * 7. HackerRank: [Cache Implementation] (<https://www.hackerrank.com/challenges/lru-cache/problem>)
- * 8. CodeChef: [Implement Cache] (<https://www.codechef.com/problems/IMCACHE>)
- * 9. 计蒜客：[LRU 缓存实现] (<https://nanti.jisuanke.com/t/41393>)
- * 10. 杭电 OJ 1816: [LRU Cache] (<http://acm.hdu.edu.cn/showproblem.php?pid=1816>)

*

* 七、补充题目（各大 OJ 平台）

- * 1. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) - 缓存优化问题
- * 2. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) - 消息缓存应用
- * 3. UVA 11525.

[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem

&problem=2520) - 缓存置换算法

- * 4. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) - 缓存查询优化
- * 5. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) - 缓存计数优化
- * 6. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) - 动态规划缓存优化
- * 7. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) - 缓存唯一性检测
- * 8. USACO Training: [Caching] (<https://train.usaco.org/>) - 缓存基础训练
- * 9. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 数据流缓存
- * 10. 赛码: [缓存设计] (<https://www.acmcoder.com/>) - 在线编程题目

*

- * 八、算法设计技巧总结
- * 1. 双向链表维护访问顺序: 最近访问的节点在尾部, 最久未访问的节点在头部
- * 2. 哈希表提供 O(1) 查找: 键到节点的直接映射
- * 3. 节点移动优化: 访问时移动到尾部, 淘汰时删除头部
- * 4. 容量控制: 当缓存满时自动淘汰最久未使用的元素
- * 5. 边界处理: 处理空缓存、单元素缓存等边界情况

*

- * 九、面试要点
- * 1. 解释 LRU 算法的核心思想和工作原理
- * 2. 分析为什么需要双向链表而不是单向链表
- * 3. 讨论哈希表在 LRU 实现中的作用
- * 4. 分析各种边界情况下的行为
- * 5. 提出线程安全实现的方案
- * 6. 讨论 LRU 算法的优缺点和适用场景

*

- * 十、工程实践中的应用场景
- * 1. 操作系统页面置换算法
- * 2. 数据库缓存管理
- * 3. Web 服务器缓存策略
- * 4. 浏览器缓存机制
- * 5. 分布式系统缓存设计
- * 6. 内存管理优化

*

- * @author 算法工程师
- * @version 1.0
- * @since 2024

*/

```
public class Code02_LRU {  
  
    // 测试链接 : https://leetcode.cn/problems/lru-cache/  
    class LRUCache {
```

```
/*
 * 双向链表节点类
 * 用于维护访问顺序，最近访问的节点在尾部，最久未访问的节点在头部
 */
class DoubleNode {
    public int key;
    public int val;
    public DoubleNode last;
    public DoubleNode next;

    public DoubleNode(int k, int v) {
        key = k;
        val = v;
    }
}

/*
 * 双向链表类
 * 提供基本的链表操作：添加节点、移动节点到尾部、删除头节点
 * 封装链表操作，简化主逻辑
 */
class DoubleList {
    private DoubleNode head;
    private DoubleNode tail;

    public DoubleList() {
        head = null;
        tail = null;
    }

    /*
     * 添加节点到链表尾部
     * 时间复杂度：O(1)
     * 关键步骤：处理空链表情况和非空链表情况
     */
    public void addNode(DoubleNode newNode) {
        if (newNode == null) {
            return;
        }
        if (head == null) {
            // 空链表情况
            head = newNode;
            tail = newNode;
        } else {
            newNode.last = tail;
            tail.next = newNode;
            tail = newNode;
        }
    }
}
```

```
        } else {
            // 非空链表情况，添加到尾部
            tail.next = newNode;
            newNode.last = tail;
            tail = newNode;
        }
    }

/*
 * 将指定节点移动到链表尾部
 * 时间复杂度: O(1)
 * 边界处理: 节点已经在尾部、节点是头节点
 */
public void moveNodeToTail(DoubleNode node) {
    // 优化: 如果节点已经在尾部, 无需操作
    if (tail == node) {
        return;
    }

    // 从原位置移除节点
    if (head == node) {
        // 节点是头节点
        head = node.next;
        head.last = null;
    } else {
        // 节点在中间位置
        node.last.next = node.next;
        node.next.last = node.last;
    }
}

// 将节点添加到尾部
node.last = tail;
node.next = null;
tail.next = node;
tail = node;
}

/*
 * 删除并返回链表头部节点 (最久未使用)
 * 时间复杂度: O(1)
 * 边界处理: 空链表、链表只有一个节点
 */
public DoubleNode removeHead() {
```

```

        if (head == null) {
            return null; // 空链表
        }
        DoubleNode ans = head;
        if (head == tail) {
            // 链表只有一个节点
            head = null;
            tail = null;
        } else {
            // 链表有多个节点
            head = ans.next;
            ans.next = null; // 断开连接, 帮助 GC
            head.last = null;
        }
        return ans;
    }

}

// 哈希表用于 O(1) 时间复杂度查找节点
private HashMap<Integer, DoubleNode> keyNodeMap;

// 双向链表维护访问顺序
private DoubleList nodeList;

// 缓存容量
private final int capacity;

/*
 * 构造函数
 * @param cap 缓存容量
 * 边界检查: 容量必须大于 0
 */
public LRUcache(int cap) {
    // 检查非法输入
    if (cap <= 0) {
        throw new IllegalArgumentException("容量必须大于 0");
    }
    keyNodeMap = new HashMap<>();
    nodeList = new DoubleList();
    capacity = cap;
}

```

```
/*
 * 获取指定 key 的值
 * @param key 键
 * @return 如果 key 存在返回对应的值，否则返回-1
 * 时间复杂度: O(1)
 * 核心逻辑: 查找节点并更新访问顺序
 */
public int get(int key) {
    if (keyNodeMap.containsKey(key)) {
        DoubleNode ans = keyNodeMap.get(key);
        // 将访问的节点移动到链表尾部 (最近访问)
        nodeList.moveToTail(ans);
        return ans.val;
    }
    return -1; // 键不存在
}

/*
 * 插入或更新键值对
 * @param key 键
 * @param value 值
 * 时间复杂度: O(1)
 * 核心逻辑: 处理更新已存在的键和插入新键两种情况
 */
public void put(int key, int value) {
    if (keyNodeMap.containsKey(key)) {
        // 更新已存在的 key
        DoubleNode node = keyNodeMap.get(key);
        node.val = value;
        // 将访问的节点移动到链表尾部 (最近访问)
        nodeList.moveToTail(node);
    } else {
        // 插入新 key
        if (keyNodeMap.size() == capacity) {
            // 缓存已满, 删除最久未使用的节点 (链表头部)
            DoubleNode removed = nodeList.removeHead();
            keyNodeMap.remove(removed.key);
        }
        // 创建新节点并添加到链表尾部和哈希表
        DoubleNode newNode = new DoubleNode(key, value);
        keyNodeMap.put(key, newNode);
        nodeList.addNode(newNode);
    }
}
```

```
        }

    }

/*
 * 补充实现：线程安全的 LRU 缓存
 * 使用读写锁实现线程安全，允许多读单写
 * 适用于读多写少的场景
 */
class ThreadSafeLRUCache {

    // 读写锁，允许多个读操作并发执行
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private final ReentrantReadWriteLock.ReadLock readLock = lock.readLock();
    private final ReentrantReadWriteLock.WriteLock writeLock = lock.writeLock();

    // 内部使用非线程安全的 LRUCache 实现
    private final LRUCache cache;

    public ThreadSafeLRUCache(int capacity) {
        this.cache = new LRUCache(capacity);
    }

    /*
     * 线程安全的 get 操作
     * 使用读锁，允许多个线程同时读取
     */
    public int get(int key) {
        readLock.lock();
        try {
            return cache.get(key);
        } finally {
            readLock.unlock();
        }
    }

    /*
     * 线程安全的 put 操作
     * 使用写锁，确保独占访问
     */
    public void put(int key, int value) {
        writeLock.lock();
        try {
            cache.put(key, value);
        } finally {

```

```

        writeLock.unlock();
    }
}

/*
 * 补充题目 1: LeetCode 146. LRU 缓存机制
 * 题目描述: 实现 LRUCache 类, 支持 get 和 put 操作, 要求 O(1) 时间复杂度
 * 与本题完全一致, 上述实现可以直接应用
 */

/*
 * 补充题目 2: 支持统计功能的 LRU 缓存扩展
 * 扩展功能: 添加命中率统计、访问次数统计等功能
 */

class EnhancedLRUCache extends LRUCache {
    private int hits = 0;      // 缓存命中次数
    private int accesses = 0; // 总访问次数
    private int evictions = 0; // 淘汰次数

    public EnhancedLRUCache(int capacity) {
        super(capacity);
    }

    @Override
    public int get(int key) {
        accesses++;
        int value = super.get(key);
        if (value != -1) {
            hits++;
        }
        return value;
    }

    // 获取命中率
    public double getHitRate() {
        return accesses == 0 ? 0 : (double) hits / accesses;
    }

    // 获取淘汰次数
    public int getEvictionCount() {
        return evictions;
    }
}

```

```
}
```

```
/**  
 * 单元测试类 - 测试 LRU 缓存的各种功能  
 */
```

```
static class LRUCacheTest {
```

```
/**  
 * 测试基本功能: 插入、查询、淘汰  
 */
```

```
public static void testBasicOperations() {  
    System.out.println("==> 测试 LRU 基本功能 ==<");  
    Code02_LRU outer = new Code02_LRU();  
    LRUCache cache = outer.new LRUCache(2);
```

```
// 测试插入和查询  
cache.put(1, 1);  
cache.put(2, 2);  
assert cache.get(1) == 1 : "插入后查询失败";  
assert cache.get(2) == 2 : "插入后查询失败";  
System.out.println("✓ 基本插入查询测试通过");
```

```
// 测试容量限制和淘汰机制  
cache.put(3, 3); // 应该淘汰键 1  
assert cache.get(1) == -1 : "淘汰机制失败";  
assert cache.get(2) == 2 : "淘汰错误键";  
assert cache.get(3) == 3 : "新插入失败";  
System.out.println("✓ 容量限制和淘汰测试通过");
```

```
// 测试访问顺序影响淘汰  
cache.get(2); // 访问键 2, 使其成为最近访问  
cache.put(4, 4); // 应该淘汰键 3  
assert cache.get(3) == -1 : "访问顺序淘汰失败";  
assert cache.get(2) == 2 : "最近访问键被错误淘汰";  
assert cache.get(4) == 4 : "新插入失败";  
System.out.println("✓ 访问顺序影响淘汰测试通过");
```

```
}
```

```
/**  
 * 测试边界情况  
 */  
public static void testEdgeCases() {  
    System.out.println("\n==> 测试边界情况 ==<");
```

```
Code02_LRU outer = new Code02_LRU();

// 测试容量为 1
LRUCache cache1 = outer.new LRUCache(1);
cache1.put(1, 1);
assert cache1.get(1) == 1 : "容量 1 插入失败";
cache1.put(2, 2);
assert cache1.get(1) == -1 : "容量 1 淘汰失败";
assert cache1.get(2) == 2 : "容量 1 新插入失败";
System.out.println("✓ 容量 1 测试通过");

// 测试空缓存查询
LRUCache cache0 = outer.new LRUCache(2);
assert cache0.get(1) == -1 : "空缓存查询失败";
System.out.println("✓ 空缓存测试通过");

// 测试更新已存在键
LRUCache cache2 = outer.new LRUCache(2);
cache2.put(1, 1);
cache2.put(1, 10); // 更新值
assert cache2.get(1) == 10 : "更新键值失败";
cache2.put(2, 2);
cache2.put(3, 3); // 应该淘汰键 2
assert cache2.get(1) == 10 : "更新后键被错误淘汰";
assert cache2.get(2) == -1 : "淘汰机制失败";
System.out.println("✓ 更新键值测试通过");
}

/**
 * 测试性能和大数据量场景
 */
public static void testPerformance() {
    System.out.println("\n==== 测试性能和大数据量 ====");
    int capacity = 1000;
    int operations = 10000;
    Code02_LRU outer = new Code02_LRU();
    LRUCache cache = outer.new LRUCache(capacity);

    long startTime = System.currentTimeMillis();

    // 批量插入
    for (int i = 0; i < operations; i++) {
        cache.put(i, i * 10);
    }
}
```

```
        if (i > capacity) {
            // 验证淘汰机制
            assert cache.get(i - capacity) == -1 : "淘汰机制失败";
        }
    }

    // 批量查询最近访问的键
    for (int i = operations - capacity; i < operations; i++) {
        int value = cache.get(i);
        assert value == i * 10 : "批量查询失败";
    }

    long endTime = System.currentTimeMillis();
    System.out.println("✓ 性能测试通过，处理 " + operations + " 次操作耗时：" + (endTime - startTime) + "ms");
}

/***
 * 测试线程安全版本
 */
public static void testThreadSafety() {
    System.out.println("\n==== 测试线程安全版本 ====");
    Code02_LRU outer = new Code02_LRU();
    ThreadSafeLRUCache threadSafeCache = outer.new ThreadSafeLRUCache(3);

    // 基本功能测试
    threadSafeCache.put(1, 100);
    threadSafeCache.put(2, 200);
    assert threadSafeCache.get(1) == 100 : "线程安全版基本功能失败";
    assert threadSafeCache.get(2) == 200 : "线程安全版基本功能失败";

    threadSafeCache.put(3, 300);
    threadSafeCache.put(4, 400); // 应该淘汰键1
    assert threadSafeCache.get(1) == -1 : "线程安全版淘汰机制失败";
    assert threadSafeCache.get(4) == 400 : "线程安全版新插入失败";

    System.out.println("✓ 线程安全版本测试通过");
}

/***
 * 运行所有测试
 */
public static void runAllTests() {
```

```
try {
    testBasicOperations();
    testEdgeCases();
    testPerformance();
    testThreadSafety();
    System.out.println("\n"+ 所有 LRU 测试通过！LRU 缓存功能正常。");
} catch (AssertionError e) {
    System.err.println("✖ LRU 测试失败：" + e.getMessage());
}
}

/**
 * 主方法 - 运行测试和演示
 */
public static void main(String[] args) {
    // 运行单元测试
    LRUCacheTest.runAllTests();

    // 演示基本功能
    System.out.println("\n==== LRU 功能演示 ====");
    Code02_LRU outer = new Code02_LRU();
    LRUCache cache = outer.new LRUCache(3);

    System.out.println("1. 插入 3 个键值对");
    cache.put(1, 10);
    cache.put(2, 20);
    cache.put(3, 30);
    System.out.println("    当前缓存：[1=10, 2=20, 3=30]");

    System.out.println("2. 访问键 1，使其成为最近使用");
    cache.get(1);
    System.out.println("    访问键 1 后，键 1 成为最近使用");

    System.out.println("3. 插入新键 4，触发淘汰机制");
    cache.put(4, 40);
    System.out.println("    插入键 4，应该淘汰最久未使用的键 2");
    System.out.println("    当前缓存：[3=30, 1=10, 4=40]");
    System.out.println("    键 2 查询结果：" + cache.get(2));

    System.out.println("\n演示完成！");
}
```

}

=====

文件: Code02_LRU.py

=====

```
# 实现 LRU 结构
```

```
,,
```

一、题目解析

LRU (Least Recently Used) 最近最少使用缓存机制是一种常用的页面置换算法。

当缓存满时，会优先淘汰最长时间未被访问的数据。

要求实现 get 和 put 操作，均要求 $O(1)$ 时间复杂度。

二、算法思路

1. 使用双向链表维护访问顺序，最近访问的节点放在尾部，最久未访问的节点在头部
2. 使用哈希表实现 $O(1)$ 时间复杂度的查找操作，映射键到节点
3. 当访问一个节点时，将其移动到链表尾部（最近访问）
4. 当插入新节点且缓存满时，删除链表头部节点（最久未访问）

三、时间复杂度分析

get 操作: $O(1)$ - 哈希表查找 + 链表节点移动

put 操作: $O(1)$ - 哈希表插入/更新 + 链表节点插入/删除

四、空间复杂度分析

$O(\text{capacity})$ - 哈希表和双向链表最多存储 capacity 个节点

五、工程化考量

1. 异常处理：检查非法输入如 $\text{capacity} \leq 0$
2. 内存管理：Python 有自动垃圾回收机制，无需手动释放内存
3. 线程安全：多线程环境下需要加锁保护
4. 可配置性：支持自定义容量
5. 性能优化：使用 Python 内置字典和自定义双向链表实现高效访问
6. 扩展性：支持添加统计功能、过期机制等扩展特性
7. 监控：在实际应用中可能需要添加性能监控指标

六、相关题目扩展

1. LeetCode 146. [LRU Cache] (<https://leetcode.com/problems/lru-cache/>) (本题原型)
2. LeetCode 460. [LFU Cache] (<https://leetcode.com/problems/lfu-cache/>) (最近最不经常使用)
3. LeetCode 432. [全 $O(1)$ 的数据结构] (<https://leetcode.com/problems/all-one-data-structure/>)
4. 牛客网：[设计 LRU 缓存结构] (<https://www.nowcoder.com/practice/e3769a5f498241bd98942db7489cbff8>)
5. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
6. LintCode 24. [LRU 缓存策略] (<https://www.lintcode.com/problem/24/>)
7. HackerRank: [Cache Implementation] (<https://www.hackerrank.com/challenges/lru-cache/problem>)

8. CodeChef: [Implement Cache] (<https://www.codechef.com/problems/IMCACHE>)
9. 计蒜客: [LRU 缓存实现] (<https://nanti.jisuanke.com/t/41393>)
10. 杭电 OJ 1816: [LRU Cache] (<http://acm.hdu.edu.cn/showproblem.php?pid=1816>)

七、Python 语言特性利用

1. 使用内置 dict 实现哈希表，性能高效
 2. 利用类的嵌套定义实现内部类封装
 3. Python 的垃圾回收自动管理内存，避免内存泄漏
 4. 在 Python 3.7+ 中，dict 保持插入顺序，可用于简化实现（但此处仍使用标准实现以保证 O(1) 操作）
- ,,,

```
class LRUCache:
    # 双向链表节点类
    # 用于维护访问顺序，最近访问的节点在尾部，最久未访问的节点在头部
    class DoubleNode:
        def __init__(self, key, val):
            self.key = key          # 键，用于在哈希表中索引
            self.val = val           # 值
            self.last = None         # 前驱节点指针
            self.next = None         # 后继节点指针

    # 双向链表类
    # 提供基本的链表操作：添加节点、移动节点到尾部、删除头节点
    # 封装链表操作，简化主逻辑
    class DoubleList:
        def __init__(self):
            self.head = None      # 链表头部指针（最久未访问）
            self.tail = None      # 链表尾部指针（最近访问）

        # 添加节点到链表尾部（最近访问）
        # 时间复杂度：O(1)
        # 关键步骤：处理空链表情况和非空链表情况
        def add_node(self, new_node):
            if new_node is None:
                return
            if self.head is None:
                # 空链表情况
                self.head = new_node
                self.tail = new_node
            else:
                # 非空链表情况，添加到尾部
                self.tail.next = new_node
                new_node.last = self.tail
                self.tail = new_node
```

```
# 将指定节点移动到链表尾部（更新为最近访问）
# 时间复杂度: O(1)
# 边界处理: 节点已经在尾部、节点是头节点
def move_node_to_tail(self, node):
    # 优化: 如果节点已经在尾部, 无需操作
    if self.tail == node:
        return

    # 从原位置移除节点
    if self.head == node:
        # 节点是头节点
        self.head = node.next
        self.head.last = None
    else:
        # 节点在中间位置
        node.last.next = node.next
        node.next.last = node.last

    # 将节点添加到尾部
    node.last = self.tail
    node.next = None
    self.tail.next = node
    self.tail = node

# 删除并返回链表头部节点（最久未使用）
# 时间复杂度: O(1)
# 边界处理: 空链表、链表只有一个节点
def remove_head(self):
    if self.head is None:
        return None # 空链表
    ans = self.head
    if self.head == self.tail:
        # 链表只有一个节点
        self.head = None
        self.tail = None
    else:
        # 链表有多个节点
        self.head = ans.next
        ans.next = None # 断开连接, 帮助垃圾回收
        self.head.last = None
    return ans
```

```
def __init__(self, capacity):
    """
    构造函数
    :param capacity: 缓存容量
    边界检查: 容量必须大于 0
    """

    # 检查非法输入
    if capacity <= 0:
        raise ValueError("容量必须大于 0")
    self.key_node_map = {} # 哈希表用于 O(1) 时间复杂度查找节点
    self.node_list = self.DoubleList() # 双向链表维护访问顺序
    self.capacity = capacity # 缓存容量

def get(self, key):
    """
    获取指定 key 的值
    :param key: 键
    :return: 如果 key 存在返回对应的值, 否则返回 -1
    时间复杂度: O(1)
    核心逻辑: 查找节点并更新访问顺序
    """

    if key in self.key_node_map:
        node = self.key_node_map[key]
        # 将访问的节点移动到链表尾部 (最近访问)
        self.node_list.move_node_to_tail(node)
        return node.val
    return -1 # 键不存在

def put(self, key, value):
    """
    插入或更新键值对
    :param key: 键
    :param value: 值
    时间复杂度: O(1)
    核心逻辑: 处理更新已存在的键和插入新键两种情况
    """

    if key in self.key_node_map:
        # 更新已存在的 key
        node = self.key_node_map[key]
        node.val = value
        # 将访问的节点移动到链表尾部 (最近访问)
        self.node_list.move_node_to_tail(node)
    else:
```

```
# 插入新 key
if len(self.key_node_map) == self.capacity:
    # 缓存已满, 删除最久未使用的节点(链表头部)
    removed = self.node_list.remove_head()
    del self.key_node_map[removed.key]
# 创建新节点并添加到链表尾部和哈希表
new_node = self.DoubleNode(key, value)
self.key_node_map[key] = new_node
self.node_list.add_node(new_node)

# 线程安全的 LRU 缓存实现
# 使用 threading.RLock 实现线程安全
class ThreadSafeLRUCache:

    def __init__(self, capacity):
        import threading
        self.lock = threading.RLock() # 可重入锁, 支持在同一线程中多次获取
        self.cache = LRUCache(capacity)

    def get(self, key):
        """
        线程安全的 get 操作
        :param key: 键
        :return: 如果 key 存在返回对应的值, 否则返回-1
        """
        with self.lock:
            return self.cache.get(key)

    def put(self, key, value):
        """
        线程安全的 put 操作
        :param key: 键
        :param value: 值
        """
        with self.lock:
            self.cache.put(key, value)

# 支持统计功能的增强版 LRU 缓存
class EnhancedLRUCache:

    def __init__(self, capacity):
        self.cache = LRUCache(capacity)
        self.hits = 0          # 缓存命中次数
        self.accesses = 0      # 总访问次数
        self.evictions = 0     # 淘汰次数
```

```
def get(self, key):
    """
    获取指定 key 的值并统计访问信息
    :param key: 键
    :return: 如果 key 存在返回对应的值, 否则返回-1
    """
    self.accesses += 1
    value = self.cache.get(key)
    if value != -1:
        self.hits += 1
    return value

def put(self, key, value):
    """
    插入或更新键值对
    :param key: 键
    :param value: 值
    """
    # 注意: 这里简化了实现, 实际需要跟踪淘汰事件
    # 可以通过修改内部 LRUCache 实现来支持淘汰回调
    self.cache.put(key, value)

def get_hit_rate(self):
    """
    获取命中率
    :return: 命中率, 取值范围[0, 1]
    """
    return 0.0 if self.accesses == 0 else self.hits / self.accesses

def print_stats(self):
    """
    打印统计信息
    """
    print(f"访问次数: {self.accesses}")
    print(f"命中次数: {self.hits}")
    print(f"命中率: {self.get_hit_rate() * 100:.2f}%")
    print(f"淘汰次数: {self.evictions}")

# 使用 Python 标准库 collections.OrderedDict 简化实现的 LRU 缓存
# 这是一种更 Pythonic 的实现方式, 但时间复杂度仍然是 O(1)
class OrderedDictLRUCache:

    def __init__(self, capacity):
```

```
if capacity <= 0:
    raise ValueError("容量必须大于 0")
from collections import OrderedDict
self.cache = OrderedDict()
self.capacity = capacity

def get(self, key):
    if key not in self.cache:
        return -1
    # 移动到末尾（最近访问）
    self.cache.move_to_end(key)
    return self.cache[key]

def put(self, key, value):
    if key in self.cache:
        # 更新并移动到末尾
        self.cache[key] = value
        self.cache.move_to_end(key)
    else:
        # 检查容量并添加新元素
        if len(self.cache) >= self.capacity:
            # 删除第一个元素（最久未访问）
            self.cache.popitem(last=False)
        self.cache[key] = value

# 测试代码
if __name__ == "__main__":
    try:
        print("== LRU Cache 基本测试 ==")
        # 创建容量为 2 的 LRU 缓存
        cache = LRUCache(2)

        # 测试用例: ["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
        #           [[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

        cache.put(1, 1) # 缓存是 {1=1}
        cache.put(2, 2) # 缓存是 {1=1, 2=2}
        print(f"get(1): {cache.get(1)}") # 返回 1, 缓存变为 {2=2, 1=1}
        cache.put(3, 3) # 该操作会使得关键字 2 作废, 缓存是 {1=1, 3=3}
        print(f"get(2): {cache.get(2)}") # 返回 -1 (未找到)
        cache.put(4, 4) # 该操作会使得关键字 1 作废, 缓存是 {4=4, 3=3}
        print(f"get(1): {cache.get(1)}") # 返回 -1 (未找到)
        print(f"get(3): {cache.get(3)}") # 返回 3, 缓存变为 {4=4, 3=3}
    except Exception as e:
        print(f"Error: {e}")

```

```

print(f"get(4): {cache.get(4)}") # 返回 4, 缓存变为 {3=3, 4=4}

# 测试增强版 LRU 缓存
print("\n==== Enhanced LRU Cache 测试 ===")
enhanced_cache = EnhancedLRUCache(3)
enhanced_cache.put(1, 1)
enhanced_cache.put(2, 2)
enhanced_cache.put(3, 3)
print(f"get(1): {enhanced_cache.get(1)}") # 命中
print(f"get(4): {enhanced_cache.get(4)}") # 未命中
enhanced_cache.put(4, 4) # 淘汰 2
print(f"get(2): {enhanced_cache.get(2)}") # 未命中
print(f"get(3): {enhanced_cache.get(3)}") # 命中
print(f"get(4): {enhanced_cache.get(4)}") # 命中

enhanced_cache.print_stats()

# 测试 Python 标准库实现的 LRU 缓存
print("\n==== OrderedDict LRU Cache 测试 ===")
ordered_cache = OrderedDictLRUCache(2)
ordered_cache.put(1, 1)
ordered_cache.put(2, 2)
print(f"get(1): {ordered_cache.get(1)}")
ordered_cache.put(3, 3)
print(f"get(2): {ordered_cache.get(2)}")
ordered_cache.put(4, 4)
print(f"get(1): {ordered_cache.get(1)}")
print(f"get(3): {ordered_cache.get(3)}")
print(f"get(4): {ordered_cache.get(4)}")

print("\n所有测试完成!")
except Exception as e:
    print(f"异常: {e}")

```

=====

文件: Code03_InsertDeleteRandom.cpp

=====

```

#include <vector>
#include <unordered_map>
#include <random>
#include <stdexcept>
#include <iostream>

```

```
// 插入、删除和获取随机元素 O(1)时间的结构
/*
 * 一、题目解析
 * 设计一个支持在平均时间复杂度 O(1)下执行以下操作的数据结构：
 * 1. insert(val)：当元素 val 不存在时返回 true，并向集合中插入该项，否则返回 false
 * 2. remove(val)：元素 val 存在时，从集合中移除该项，返回 true，否则返回 false
 * 3. getRandom：随机返回现有集合中的一项，每个元素应该有相同的概率被返回
 *
 * 二、算法思路
 * 1. 使用数组(vector)存储元素，实现 O(1)时间复杂度的随机访问
 * 2. 使用哈希表(unordered_map)存储元素值到其在数组中索引的映射，实现 O(1)时间复杂度的查找
 * 3. 插入操作：直接在数组末尾添加元素，并在哈希表中记录其索引
 * 4. 删除操作：将要删除的元素与数组末尾元素交换，然后删除末尾元素，更新哈希表
 * 5. 随机获取：使用随机数生成器随机生成索引，访问数组中对应元素
 *
 * 三、时间复杂度分析
 * insert 操作：O(1) - 数组末尾插入 + 哈希表插入
 * remove 操作：O(1) - 哈希表查找 + 数组元素交换 + 数组末尾删除 + 哈希表更新
 * getRandom 操作：O(1) - 随机索引生成 + 数组访问
 *
 * 四、空间复杂度分析
 * O(n) - n 为集合中元素个数，需要数组和哈希表分别存储元素和索引映射
 *
 * 五、工程化考量
 * 1. 异常处理：处理空集合的 getRandom 操作
 * 2. 边界场景：空集合、单元素集合等
 * 3. 随机性：确保 getRandom 方法能真正等概率返回每个元素
 * 4. 内存管理：C++需要手动管理内存
 *
 * 六、相关题目扩展
 * 1. LeetCode 380. 常数时间插入、删除和获取随机元素（本题）
 * 2. LeetCode 381. 常数时间插入、删除和获取随机元素-允许重复
 */
```

```
class RandomizedSet {
private:
    // 哈希表存储元素值到其在数组中索引的映射
    std::unordered_map<int, int> map;

    // 数组存储元素值
    std::vector<int> arr;
```

```
// 随机数生成器
std::random_device rd;
std::mt19937 gen;

public:
    // 构造函数
    RandomizedSet() : gen(rd()) {}

    /*
     * 插入元素
     * @param val 要插入的元素
     * @return 如果元素不存在则插入并返回 true, 否则返回 false
     * 时间复杂度: O(1)
     */
    bool insert(int val) {
        // 检查元素是否已存在
        if (map.find(val) != map.end()) {
            return false;
        }
        // 在数组末尾添加元素
        map[val] = arr.size();
        arr.push_back(val);
        return true;
    }

    /*
     * 删除元素
     * @param val 要删除的元素
     * @return 如果元素存在则删除并返回 true, 否则返回 false
     * 时间复杂度: O(1)
     */
    bool remove(int val) {
        // 检查元素是否存在
        if (map.find(val) == map.end()) {
            return false;
        }
        // 获取要删除元素的索引
        int valIndex = map[val];
        // 获取数组末尾元素的值
        int endValue = arr.back();
        // 将末尾元素放到要删除元素的位置
        map[endValue] = valIndex;
        arr[valIndex] = endValue;
    }
}
```

```

    // 删除末尾元素
    map.erase(val);
    arr.pop_back();
    return true;
}

/*
 * 随机获取元素
 * @return 随机返回集合中的一个元素
 * 时间复杂度: O(1)
 */
int getRandom() {
    // 检查集合是否为空
    if (arr.empty()) {
        throw std::runtime_error("集合为空，无法获取随机元素");
    }
    // 随机返回数组中的一个元素
    std::uniform_int_distribution<> dis(0, arr.size() - 1);
    return arr[dis(gen)];
}
};

// 测试代码
int main() {
    RandomizedSet randomizedSet;

    // 测试用例: ["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]
    //           [[], [1], [2], [2], [], [1], [2], []]

    std::cout << std::boolalpha; // 使 bool 值输出为 true/false 而不是 1/0
    std::cout << "insert(1): " << randomizedSet.insert(1) << std::endl; // true
    std::cout << "remove(2): " << randomizedSet.remove(2) << std::endl; // false
    std::cout << "insert(2): " << randomizedSet.insert(2) << std::endl; // true
    std::cout << "getRandom: " << randomizedSet.getRandom() << std::endl; // 1 或 2
    std::cout << "remove(1): " << randomizedSet.remove(1) << std::endl; // true
    std::cout << "insert(2): " << randomizedSet.insert(2) << std::endl; // false
    std::cout << "getRandom: " << randomizedSet.getRandom() << std::endl; // 2

    return 0;
}
=====
```

文件: Code03_InsertDeleteRandom.java

```
=====
```

```
package class035;
```

```
import java.util.*;
```

```
/**
```

```
* 实现支持 O(1) 时间复杂度的插入、删除和随机获取元素的数据结构
```

```
* 题目来源: LeetCode 380. Insert Delete GetRandom O(1)
```

```
* 网址: https://leetcode.com/problems/insert-delete-getrandom-o1/
```

```
*
```

```
* 一、题目解析
```

```
* 实现一个支持以下操作的数据结构, 所有操作的时间复杂度都要求为 O(1):
```

```
* 1. insert(val): 插入元素, 如果元素不存在则插入并返回 true, 否则返回 false
```

```
* 2. remove(val): 删除元素, 如果元素存在则删除并返回 true, 否则返回 false
```

```
* 3. getRandom(): 随机返回集合中的一个元素, 每个元素被返回的概率相同
```

```
*
```

```
* 二、算法思路
```

```
* 1. 使用动态数组(ArrayList)存储元素, 支持 O(1) 随机访问
```

```
* 2. 使用哈希表(HashMap)维护元素到索引的映射, 支持 O(1) 查找
```

```
* 3. 插入操作: 直接在数组末尾添加元素, 并在哈希表中记录索引
```

```
* 4. 删除操作: 将要删除元素与数组末尾元素交换, 然后删除末尾元素, 更新哈希表
```

```
* 5. 随机获取: 使用 Random 类生成随机索引, 直接访问数组元素
```

```
*
```

```
* 三、时间复杂度分析
```

```
* - insert(val): O(1) 平均时间复杂度
```

```
* - remove(val): O(1) 平均时间复杂度
```

```
* - getRandom(): O(1) 时间复杂度
```

```
*
```

```
* 四、空间复杂度分析
```

```
* O(n), 其中 n 是存储的元素数量, 需要数组和哈希表存储所有元素
```

```
*
```

```
* 五、工程化考量
```

```
* 1. 异常处理: 处理空集合的 getRandom 操作
```

```
* 2. 边界情况: 插入重复元素、删除不存在元素、空集合操作
```

```
* 3. 内存管理: Java 自动垃圾回收, 但仍需注意大对象的内存消耗
```

```
* 4. 线程安全: 当前实现非线程安全, 如需线程安全可使用 Collections.synchronizedList 等
```

```
* 5. 性能优化: 利用数组末尾操作的 O(1) 特性优化删除操作
```

```
* 6. 可扩展性: 可扩展为支持泛型的通用数据结构
```

```
*
```

```
* 六、相关题目扩展
```

```
* 1. LeetCode 380. [Insert Delete GetRandom O(1)] (https://leetcode.com/problems/insert-delete-
```

getrandom=01/）（本题）

- * 2. LeetCode 381. [Insert Delete GetRandom O(1) – Duplicates allowed] (<https://leetcode.com/problems/insert-delete-getrandom-o1-duplicates-allowed/>) （允许重复元素）
- * 3. 牛客网：[设计支持 O(1) 插入删除和随机访问的数据结构] (https://www.nowcoder.com/practice/11165e95382547_cab9b6518e2760384d)
- * 4. 剑指 Offer II 030. [插入、删除和随机访问都是 O(1) 的容器] (<https://leetcode.cn/problems/FortPu/>)
- * 5. LintCode 657. [Insert Delete GetRandom O(1)] (<https://www.lintcode.com/problem/657/>)
- * 6. HackerRank: [Data Structures – RandomizedSet] (<https://www.hackerrank.com/challenges/java-hashset/problem>)
- * 7. CodeChef: [Random Set Operations] (<https://www.codechef.com/problems/RANDSET>)
- * 8. 计蒜客：[O(1) 数据结构] (<https://nanti.jisuanke.com/t/41394>)
- *
- * 七、补充题目（各大 OJ 平台）
 - * 1. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) – 集合操作优化
 - * 2. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) – 消息集合处理
 - * 3. UVA 11525.
[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2520) – 集合排列问题
 - * 4. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) – 集合查询优化
 - * 5. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) – 集合计数优化
 - * 6. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) – 动态规划集合优化
 - * 7. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) – 集合唯一性检测
 - * 8. USACO Training: [Set Operations] (<https://train.usaco.org/>) – 集合基础训练
 - * 9. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) – 数据流集合
 - * 10. 赛码：[集合设计] (<https://www.acmcoder.com/>) – 在线编程题目
 - *
- * 八、算法设计技巧总结
 - * 1. 数组+哈希表组合：利用数组的 O(1) 随机访问和哈希表的 O(1) 查找
 - * 2. 交换删除法：通过将要删除元素与末尾元素交换来实现 O(1) 删除
 - * 3. 索引映射维护：哈希表维护元素到索引的映射，确保操作一致性
 - * 4. 边界优化：特殊处理数组末尾操作，避免不必要的元素移动
 - * 5. 随机均匀性：使用标准随机数生成器保证元素返回概率相等
 - *
- * 九、面试要点
 - * 1. 解释为什么需要数组和哈希表的组合
 - * 2. 分析删除操作中交换元素的必要性
 - * 3. 讨论各种边界情况的处理

```
* 4. 分析时间复杂度和空间复杂度
* 5. 提出可能的扩展和优化方向
* 6. 讨论线程安全性问题和解决方案
*
* 十、工程实践中的应用场景
* 1. 随机抽样系统
* 2. 负载均衡器中的服务器管理
* 3. 缓存系统中的键管理
* 4. 游戏开发中的道具管理
* 5. 数据库索引优化
* 6. 推荐系统中的候选集维护
*/
public class Code03_InsertDeleteRandom {

    private Map<Integer, Integer> valueToIndex; // 值到索引的映射，用于 O(1) 查找
    private List<Integer> values; // 存储值的列表，用于 O(1) 随机访问
    private Random random; // 随机数生成器，用于 O(1) 随机选择

    /**
     * 初始化数据结构 */
    public Code03_InsertDeleteRandom() {
        valueToIndex = new HashMap<>();
        values = new ArrayList<>();
        random = new Random();
    }

    /**
     * 插入元素
     * @param val 要插入的值
     * @return 如果值不存在则插入成功返回 true，否则返回 false
     * 时间复杂度：O(1) 平均时间复杂度
     * 空间复杂度：O(1)
     */
    public boolean insert(int val) {
        // 检查元素是否已存在
        if (valueToIndex.containsKey(val)) {
            return false;
        }
        // 在哈希表中记录值到索引的映射
        valueToIndex.put(val, values.size());
        // 在数组末尾添加值
        values.add(val);
        return true;
    }
}
```

```
/**  
 * 删除元素  
 * @param val 要删除的值  
 * @return 如果值存在则删除成功返回 true, 否则返回 false  
 * 时间复杂度: O(1) 平均时间复杂度  
 * 空间复杂度: O(1)  
 * 核心思想: 将要删除的元素与数组末尾元素交换, 然后删除末尾元素  
 */  
public boolean remove(int val) {  
    // 检查元素是否存在  
    if (!valueToIndex.containsKey(val)) {  
        return false;  
    }  
  
    // 获取要删除元素的索引  
    int index = valueToIndex.get(val);  
    // 获取数组末尾元素  
    int lastElement = values.get(values.size() - 1);  
  
    // 将末尾元素移动到要删除的位置  
    values.set(index, lastElement);  
    // 更新末尾元素在哈希表中的索引  
    valueToIndex.put(lastElement, index);  
  
    // 删除数组末尾元素  
    values.remove(values.size() - 1);  
    // 从哈希表中删除该元素  
    valueToIndex.remove(val);  
  
    return true;  
}  
  
/**  
 * 随机获取一个元素  
 * @return 随机元素  
 * 时间复杂度: O(1)  
 * 空间复杂度: O(1)  
 * 核心思想: 使用 Random 类生成随机索引, 直接访问数组元素  
 */  
public int getRandom() {  
    // 生成 0 到 size-1 的随机索引  
    int randomIndex = random.nextInt(values.size());
```

```
// 返回对应索引的元素
return values.get(randomIndex);
}

// ===== 单元测试和功能演示 =====

/**
 * 单元测试类 - 测试 RandomizedSet 的各种功能
 */
public static class RandomizedSetTest {

    /**
     * 测试边界情况
     */
    public static void testEdgeCases() {
        System.out.println("\n==== 测试边界情况 ===");

        Code03_InsertDeleteRandom set = new Code03_InsertDeleteRandom();

        // 测试空集合
        assert !set.remove(1) : "空集合删除应该返回 false";
        try {
            set.getRandom();
            assert false : "空集合 getRandom 应该抛出异常";
        } catch (Exception e) {
            System.out.println("✓ 空集合异常处理正确");
        }

        // 测试插入重复元素
        assert set.insert(1) : "第一次插入 1 应该成功";
        assert !set.insert(1) : "第二次插入 1 应该失败";
        System.out.println("✓ 重复插入测试通过");

        // 测试删除不存在的元素
        assert !set.remove(999) : "删除不存在的元素应该返回 false";
        System.out.println("✓ 删除不存在元素测试通过");

        // 测试插入删除后 getRandom
        set.insert(2);
        set.insert(3);
        set.remove(2);

        // 验证删除后集合状态
    }
}
```

```
assert set.insert(2) : "删除后重新插入应该成功";
assert set.getRandom() != 999 : "getRandom 应该返回有效值";
System.out.println("✓ 删除后状态测试通过");
}

/**
 * 测试性能和大数据量
 */
public static void testPerformance() {
    System.out.println("\n==== 测试性能和大数据量 ===");

    Code03_InsertDeleteRandom set = new Code03_InsertDeleteRandom();
    int n = 10000;
    long startTime = System.currentTimeMillis();

    // 批量插入
    for (int i = 0; i < n; i++) {
        set.insert(i);
    }

    // 批量删除
    for (int i = 0; i < n; i += 2) {
        set.remove(i);
    }

    // 随机操作混合
    for (int i = 0; i < n; i++) {
        if (i % 3 == 0) {
            set.insert(i + n);
        } else if (i % 5 == 0) {
            set.remove(i);
        } else {
            set.getRandom();
        }
    }

    long endTime = System.currentTimeMillis();
    System.out.println("✓ 性能测试通过, 处理 " + n + " 次操作耗时: " + (endTime - startTime) + "ms");
}

/**
 * 测试随机性分布

```

```

*/
public static void testRandomness() {
    System.out.println("\n==== 测试随机性分布 ===");

    Code03_InsertDeleteRandom set = new Code03_InsertDeleteRandom();
    int[] testValues = {1, 2, 3, 4, 5};

    // 插入测试数据
    for (int val : testValues) {
        set.insert(val);
    }

    // 统计随机分布
    int[] count = new int[6]; // 索引 1-5 对应值 1-5
    int trials = 10000;

    for (int i = 0; i < trials; i++) {
        int randomVal = set.getRandom();
        count[randomVal]++;
    }

    // 验证分布均匀性（每个值应该出现约 2000 次）
    double expected = trials / 5.0;
    double tolerance = expected * 0.1; // 10%容差

    for (int i = 1; i <= 5; i++) {
        double frequency = count[i] / (double)trials;
        assert Math.abs(count[i] - expected) < tolerance :
            "值" + i + "出现频率不均匀: " + count[i] + " vs " + expected;
    }
    System.out.println("✓ 随机性分布测试通过");
}

/**
 * 运行所有测试
 */
public static void runAllTests() {
    try {
        testEdgeCases();
        testPerformance();
        testRandomness();
        System.out.println("\n🎉 所有 RandomizedSet 测试通过！功能正常。");
    } catch (AssertionError e) {

```

```
        System.out.println("X 测试失败: " + e.getMessage());
    }
}

/**
 * 功能演示
 */
public static void demonstrate() {
    System.out.println("\n==== RandomizedSet 功能演示 ===");

    Code03_InsertDeleteRandom set = new Code03_InsertDeleteRandom();

    System.out.println("1. 插入元素 1, 2, 3");
    set.insert(1);
    set.insert(2);
    set.insert(3);

    System.out.println("2. 尝试插入重复元素 2: " + set.insert(2));

    System.out.println("3. 删除元素 2: " + set.remove(2));
    System.out.println("4. 再次删除元素 2: " + set.remove(2));

    System.out.println("5. 随机获取元素:");
    for (int i = 0; i < 5; i++) {
        System.out.println(" 第" + (i+1) + "次随机: " + set.getRandom());
    }

    System.out.println("6. 插入元素 4, 5");
    set.insert(4);
    set.insert(5);

    System.out.println("7. 最终随机抽样:");
    for (int i = 0; i < 3; i++) {
        System.out.println(" 随机值: " + set.getRandom());
    }

    System.out.println("\n演示完成!");
}

/**
 * 主函数 - 运行测试和演示
 */

```

```
public static void main(String[] args) {  
    // 运行单元测试  
    RandomizedSetTest.runAllTests();  
  
    // 功能演示  
    demonstrate();  
}  
}
```

文件: Code03_InsertDeleteRandom.py

```
import random  
  
# 插入、删除和获取随机元素 O(1)时间的结构  
,,,
```

一、题目解析

设计一个支持在平均时间复杂度 $O(1)$ 下执行以下操作的数据结构:

1. insert(val): 当元素 val 不存在时返回 true，并向集合中插入该项，否则返回 false
2. remove(val): 元素 val 存在时，从集合中移除该项，返回 true，否则返回 false
3. getRandom: 随机返回现有集合中的一项，每个元素应该有相同的概率被返回

二、算法思路

1. 使用数组(list)存储元素，实现 $O(1)$ 时间复杂度的随机访问
2. 使用哈希表(dict)存储元素值到其在数组中索引的映射，实现 $O(1)$ 时间复杂度的查找
3. 插入操作：直接在数组末尾添加元素，并在哈希表中记录其索引
4. 删除操作：将要删除的元素与数组末尾元素交换，然后删除末尾元素，更新哈希表
5. 随机获取：使用 random 模块随机生成索引，访问数组中对应元素

三、时间复杂度分析

insert 操作: $O(1)$ - 数组末尾插入 + 哈希表插入

remove 操作: $O(1)$ - 哈希表查找 + 数组元素交换 + 数组末尾删除 + 哈希表更新

getRandom 操作: $O(1)$ - 随机索引生成 + 数组访问

四、空间复杂度分析

$O(n)$ - n 为集合中元素个数，需要数组和哈希表分别存储元素和索引映射

五、工程化考量

1. 异常处理：处理空集合的 getRandom 操作
2. 边界场景：空集合、单元素集合等
3. 随机性：确保 getRandom 方法能真正等概率返回每个元素
4. 内存管理：Python 有自动垃圾回收机制

六、相关题目扩展

1. LeetCode 380. 常数时间插入、删除和获取随机元素（本题）
2. LeetCode 381. 常数时间插入、删除和获取随机元素-允许重复
- ...

```
class RandomizedSet:  
    def __init__(self):  
        """构造函数"""  
        self.map = {} # 哈希表存储元素值到其在数组中索引的映射  
        self.arr = [] # 数组存储元素值  
  
    def insert(self, val: int) -> bool:  
        """  
        插入元素  
        :param val: 要插入的元素  
        :return: 如果元素不存在则插入并返回 True, 否则返回 False  
        时间复杂度: O(1)  
        """  
  
        # 检查元素是否已存在  
        if val in self.map:  
            return False  
        # 在数组末尾添加元素  
        self.map[val] = len(self.arr)  
        self.arr.append(val)  
        return True  
  
    def remove(self, val: int) -> bool:  
        """  
        删除元素  
        :param val: 要删除的元素  
        :return: 如果元素存在则删除并返回 True, 否则返回 False  
        时间复杂度: O(1)  
        """  
  
        # 检查元素是否存在  
        if val not in self.map:  
            return False  
        # 获取要删除元素的索引  
        val_index = self.map[val]  
        # 获取数组末尾元素的值  
        end_value = self.arr[-1]  
        # 将末尾元素放到要删除元素的位置  
        self.map[end_value] = val_index
```

```

        self.arr[val_index] = end_value
        # 删除末尾元素
        del self.map[val]
        self.arr.pop()
        return True

    def getRandom(self) -> int:
        """
        随机获取元素
        :return: 随机返回集合中的一个元素
        时间复杂度: O(1)
        """
        # 检查集合是否为空
        if not self.arr:
            raise Exception("集合为空，无法获取随机元素")
        # 随机返回数组中的一个元素
        return random.choice(self.arr)

# 测试代码
if __name__ == "__main__":
    randomizedSet = RandomizedSet()

    # 测试用例: ["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]
    #           [[], [1], [2], [2], [], [1], [2], []]

    print("insert(1):", randomizedSet.insert(1))      # True
    print("remove(2):", randomizedSet.remove(2))      # False
    print("insert(2):", randomizedSet.insert(2))      # True
    print("getRandom:", randomizedSet.getRandom())     # 1 或 2
    print("remove(1):", randomizedSet.remove(1))      # True
    print("insert(2):", randomizedSet.insert(2))      # False
    print("getRandom:", randomizedSet.getRandom())     # 2

```

=====

文件: Code04_InsertDeleteRandomDuplicatesAllowed.cpp

=====

```

#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <random>
#include <stdexcept>

```

```
#include <iostream>

// 插入、删除和获取随机元素 O(1) 时间且允许有重复数字的结构
/*
 * 一、题目解析
 * 设计一个支持在平均时间复杂度 O(1) 下执行以下操作的数据结构（允许重复元素）：
 * 1. insert(val)：将一个元素 val 插入到集合中，返回 true
 * 2. remove(val)：如果元素 val 存在，则从中删除一个实例，返回 true，否则返回 false
 * 3. getRandom：随机返回集合中的一个元素，每个元素被返回的概率与其在集合中的数量成线性关系
 *
 * 二、算法思路
 * 与不允许重复的版本相比，主要变化在于需要处理重复元素：
 * 1. 使用数组(vector)存储所有元素，实现 O(1) 时间复杂度的随机访问
 * 2. 使用哈希表(unordered_map)存储元素值到其在数组中索引集合的映射
 * 3. 插入操作：在数组末尾添加元素，并在哈希表中记录其索引
 * 4. 删除操作：将要删除的元素与数组末尾元素交换，然后删除末尾元素，更新哈希表
 * 5. 随机获取：使用随机数生成器随机生成索引，访问数组中对应元素
 *
 * 三、时间复杂度分析
 * insert 操作：O(1) - 数组末尾插入 + 哈希表更新
 * remove 操作：O(1) - 哈希表查找 + 数组元素交换 + 数组末尾删除 + 哈希表更新
 * getRandom 操作：O(1) - 随机索引生成 + 数组访问
 *
 * 四、空间复杂度分析
 * O(n) - n 为集合中元素个数，需要数组和哈希表分别存储元素和索引映射
 *
 * 五、工程化考量
 * 1. 异常处理：处理空集合的 getRandom 操作
 * 2. 边界场景：空集合、单元素集合等
 * 3. 随机性：确保 getRandom 方法能真正按概率返回每个元素
 * 4. 内存管理：C++ 需要手动管理内存
 *
 * 六、相关题目扩展
 * 1. LeetCode 381. 常数时间插入、删除和获取随机元素-允许重复（本题）
 * 2. LeetCode 380. 常数时间插入、删除和获取随机元素
 * 3. 牛客网相关题目
 */


```

```
class RandomizedCollection {
private:
    // 哈希表存储元素值到其在数组中索引集合的映射
    std::unordered_map<int, std::unordered_set<int>> map;
```

```
// 数组存储所有元素值
std::vector<int> arr;

// 随机数生成器
std::random_device rd;
std::mt19937 gen;

public:
    // 构造函数
    RandomizedCollection() : gen(rd()) {}

    /*
     * 插入元素
     * @param val 要插入的元素
     * @return 总是返回 true
     * 时间复杂度: O(1)
     */
    bool insert(int val) {
        // 在数组末尾添加元素
        arr.push_back(val);
        // 获取或创建该元素值对应的索引集合
        std::unordered_set<int>& set = map[val];
        // 将新索引添加到集合中
        set.insert(arr.size() - 1);
        // 当且仅当该元素第一次插入时返回 true
        return set.size() == 1;
    }

    /*
     * 删除元素
     * @param val 要删除的元素
     * @return 如果元素存在则删除并返回 true，否则返回 false
     * 时间复杂度: O(1)
     */
    bool remove(int val) {
        // 检查元素是否存在
        if (map.find(val) == map.end()) {
            return false;
        }
        // 获取该元素值对应的索引集合
        std::unordered_set<int>& valSet = map[val];
        // 获取其中一个索引（任意一个）
        int valAnyIndex = *(valSet.begin());
```

```
// 获取数组末尾元素的值
int endValue = arr.back();
// 如果要删除的元素就是末尾元素
if (val == endValue) {
    // 直接从索引集合中删除该索引
    valSet.erase(arr.size() - 1);
} else {
    // 获取末尾元素值对应的索引集合
    std::unordered_set<int>& endValueSet = map[val];
    // 将末尾元素的索引更新为要删除元素的索引
    endValueSet.insert(valAnyIndex);
    // 更新数组中要删除元素位置的值为末尾元素值
    arr[valAnyIndex] = endValue;
    // 从末尾元素的索引集合中删除原末尾索引
    endValueSet.erase(arr.size() - 1);
    // 从要删除元素的索引集合中删除该索引
    valSet.erase(valAnyIndex);
}
// 删除数组末尾元素
arr.pop_back();
// 如果要删除元素的索引集合为空，则从哈希表中删除该元素
if (valSet.empty()) {
    map.erase(val);
}
return true;
}

/*
 * 随机获取元素
 * @return 随机返回集合中的一个元素
 * 时间复杂度: O(1)
 */
int getRandom() {
    // 检查集合是否为空
    if (arr.empty()) {
        throw std::runtime_error("集合为空，无法获取随机元素");
    }
    // 随机返回数组中的一个元素
    std::uniform_int_distribution<int> dis(0, arr.size() - 1);
    return arr[dis(gen)];
}
};
```

```

// 测试代码
int main() {
    RandomizedCollection collection;

    // 简单测试
    std::cout << std::boolalpha;
    std::cout << "insert(1): " << collection.insert(1) << std::endl; // true
    std::cout << "insert(1): " << collection.insert(1) << std::endl; // false
    std::cout << "insert(2): " << collection.insert(2) << std::endl; // true
    std::cout << "remove(1): " << collection.remove(1) << std::endl; // true
    std::cout << "getRandom: " << collection.getRandom() << std::endl; // 1 or 2

    return 0;
}
=====
```

文件: Code04_InsertDeleteRandomDuplicatesAllowed.java

```
=====
package class035;

import java.util.*;

/**
 * 实现支持重复元素的 O(1) 时间复杂度插入、删除和随机获取元素的数据结构
 * 题目来源: LeetCode 381. Insert Delete GetRandom O(1) - Duplicates allowed
 * 网址: https://leetcode.com/problems/insert-delete-getrandom-o1-duplicates-allowed/
 *
 * 一、题目解析
 * 实现一个支持以下操作的数据结构，所有操作的时间复杂度都要求为 O(1):
 * 1. insert(val): 插入元素，允许重复元素，总是返回 true
 * 2. remove(val): 删除元素的一个实例，如果元素存在则删除并返回 true，否则返回 false
 * 3. getRandom(): 随机返回集合中的一个元素，每个元素被返回的概率与其在集合中的数量成正比
 *
 * 二、算法思路
 * 1. 使用动态数组(ArrayList)存储所有元素，支持 O(1) 随机访问
 * 2. 使用哈希表(Map<Integer, Set<Integer>>)维护元素到索引集合的映射，支持 O(1) 查找
 * 3. 插入操作：在数组末尾添加元素，并在哈希表中记录该元素对应的所有索引
 * 4. 删除操作：找到要删除元素的任意一个索引，将其与数组末尾元素交换，然后删除末尾元素，更新哈希表
 *
 * 5. 随机获取：使用 Random 类生成随机索引，直接访问数组元素
 *
 * 三、时间复杂度分析

```

* - insert(val): O(1) 平均时间复杂度

* - remove(val): O(1) 平均时间复杂度

* - getRandom(): O(1) 时间复杂度

*

* 四、空间复杂度分析

* O(n), 其中 n 是存储的元素数量, 需要数组和哈希表存储所有元素

*

* 五、工程化考量

* 1. 异常处理: 处理空集合的 getRandom 操作

* 2. 边界情况: 插入大量重复元素、删除所有实例、空集合操作

* 3. 内存管理: Java 自动垃圾回收, 但仍需注意大对象的内存消耗

* 4. 线程安全: 当前实现非线程安全, 如需线程安全可使用 Collections.synchronizedList 等

* 5. 性能优化: 利用 LinkedHashSet 保证索引集合的有序性, 优化删除操作

* 6. 可扩展性: 可扩展为支持泛型的通用数据结构

*

* 六、相关题目扩展

* 1. LeetCode 381. [Insert Delete GetRandom O(1) – Duplicates allowed] (<https://leetcode.com/problems/insert-delete-getrandom-o1-duplicates-allowed/>) (本题)

* 2. LeetCode 380. [Insert Delete GetRandom O(1)] (<https://leetcode.com/problems/insert-delete-getrandom-o1/>) (不允许重复元素)

* 3. 牛客网: [设计支持重复元素的 O(1) 数据结

构] (<https://www.nowcoder.com/practice/11165e95382547cab9b6518e2760384d>)

* 4. 剑指 Offer II 030. [插入、删除和随机访问都是 O(1) 的容
器] (<https://leetcode.cn/problems/FortPu/>)

* 5. LintCode 657. [Insert Delete GetRandom O(1)] (<https://www.lintcode.com/problem/657/>)

* 6. HackerRank: [Data Structures – RandomizedSet with
Duplicates] (<https://www.hackerrank.com/challenges/java-hashset/problem>)

* 7. CodeChef: [Random Set Operations with
Duplicates] (<https://www.codechef.com/problems/RANDSET>)

* 8. 计蒜客: [O(1) 数据结构 (允许重复)] (<https://nanti.jisuanke.com/t/41394>)

*

* 七、补充题目 (各大 OJ 平台)

* 1. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) – 集合操
作优化

* 2. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) – 消息集合处理

* 3. UVA 11525.

[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2520) – 集合排列问题

* 4. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) – 集合查询优化

* 5. Project Euler 543. [Counting the Number of Close
Pairs] (<https://projecteuler.net/problem=543>) – 集合计数优化

* 6. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) – 动态规划集合优

化

- * 7. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) - 集合唯一性检测
- * 8. USACO Training: [Set Operations with Duplicates] (<https://train.usaco.org/>) - 集合基础训练
- * 9. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 数据流集合
- * 10. 赛码: [集合设计 (允许重复)] (<https://www.acmCoder.com/>) - 在线编程题目

*

* 八、算法设计技巧总结

- * 1. 数组+哈希表组合: 利用数组的 O(1) 随机访问和哈希表的 O(1) 查找
- * 2. 索引集合维护: 使用 Set 存储元素的所有索引, 支持快速查找和删除
- * 3. 交换删除法: 通过将要删除元素与末尾元素交换来实现 O(1) 删除
- * 4. 边界优化: 特殊处理数组末尾操作, 避免不必要的元素移动
- * 5. 随机均匀性: 使用标准随机数生成器保证元素返回概率与其数量成正比

*

* 九、面试要点

- * 1. 解释与不允许重复元素版本的区别和实现差异
- * 2. 分析删除操作中索引集合的维护策略
- * 3. 讨论各种边界情况的处理
- * 4. 分析时间复杂度和空间复杂度
- * 5. 提出可能的扩展和优化方向
- * 6. 讨论线程安全性问题和解决方案

*

* 十、工程实践中的应用场景

- * 1. 随机抽样系统 (支持重复元素)
- * 2. 负载均衡器中的服务器管理 (支持多实例)
- * 3. 缓存系统中的键管理 (支持重复键)
- * 4. 游戏开发中的道具管理 (支持重复道具)
- * 5. 数据库索引优化 (支持重复值)
- * 6. 推荐系统中的候选集维护 (支持重复推荐)

*/

```
public class Code04_InsertDeleteRandomDuplicatesAllowed {
```

```
    private Map<Integer, Set<Integer>> valueToIndices; // 值到索引集合的映射, 用于 O(1) 查找
    private List<Integer> values; // 存储值的列表, 用于 O(1) 随机访问
    private Random random; // 随机数生成器, 用于 O(1) 随机选择
```

```
    /** 初始化数据结构 */
```

```
    public Code04_InsertDeleteRandomDuplicatesAllowed() {
        valueToIndices = new HashMap<>();
        values = new ArrayList<>();
        random = new Random();
    }
```

```
    /**
```

```
* 插入元素（允许重复）
* @param val 要插入的值
* @return 总是返回 true，因为允许重复
* 时间复杂度：O(1) 平均时间复杂度
* 空间复杂度：O(1)
*/
public boolean insert(int val) {
    // 如果值不存在，创建新的索引集合
    if (!valueToIndices.containsKey(val)) {
        valueToIndices.put(val, new LinkedHashSet<>());
    }

    // 添加新索引
    valueToIndices.get(val).add(values.size());
    values.add(val);

    return true;
}

/**
* 删除元素
* @param val 要删除的值
* @return 如果值存在则删除成功返回 true，否则返回 false
* 时间复杂度：O(1) 平均时间复杂度
* 空间复杂度：O(1)
* 核心思想：找到要删除元素的任意一个索引，将其与数组末尾元素交换，然后删除末尾元素
*/
public boolean remove(int val) {
    // 检查元素是否存在
    if (!valueToIndices.containsKey(val) || valueToIndices.get(val).isEmpty()) {
        return false;
    }

    // 获取要删除的值的任意一个索引
    int removeIndex = valueToIndices.get(val).iterator().next();
    int lastIndex = values.size() - 1;
    int lastElement = values.get(lastIndex);

    // 如果删除的不是最后一个元素，需要交换
    if (removeIndex != lastIndex) {
        // 将最后一个元素移动到要删除的位置
        values.set(removeIndex, lastElement);
    }
}
```

```
// 更新最后一个元素的索引映射
Set<Integer> lastElementIndices = valueToIndices.get(lastElement);
if (lastElementIndices != null) {
    lastElementIndices.remove(lastIndex);
    lastElementIndices.add(removeIndex);
}

}

// 删除要删除的值的索引
Set<Integer> valIndices = valueToIndices.get(val);
valIndices.remove(removeIndex);

// 删除最后一个元素
values.remove(lastIndex);

// 如果值的索引集合为空，删除该键
if (valIndices.isEmpty()) {
    valueToIndices.remove(val);
}

return true;
}

/***
 * 随机获取一个元素
 * @return 随机元素
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 * 核心思想: 使用 Random 类生成随机索引，直接访问数组元素
 * 概率特性: 每个元素被返回的概率与其在集合中的数量成正比
 */
public int getRandom() {
    // 生成 0 到 size-1 的随机索引
    int randomIndex = random.nextInt(values.size());
    // 返回对应索引的元素
    return values.get(randomIndex);
}

// ===== 单元测试和功能演示 =====

/***
 * 单元测试类 - 测试 RandomizedCollection 的各种功能
 */
```

```
public static class RandomizedCollectionTest {  
  
    /**  
     * 测试边界情况  
     */  
    public static void testEdgeCases() {  
        System.out.println("\n==== 测试边界情况 ====");  
  
        Code04_InsertDeleteRandomDuplicatesAllowed collection = new  
Code04_InsertDeleteRandomDuplicatesAllowed();  
  
        // 测试空集合  
        assert !collection.remove(1) : "空集合删除应该返回 false";  
        try {  
            collection.getRandom();  
            assert false : "空集合 getRandom 应该抛出异常";  
        } catch (Exception e) {  
            System.out.println("✓ 空集合异常处理正确");  
        }  
  
        // 测试插入重复元素  
        assert collection.insert(1) : "第一次插入 1 应该成功";  
        assert collection.insert(1) : "第二次插入 1 应该成功 (允许重复)";  
        assert collection.insert(1) : "第三次插入 1 应该成功 (允许重复)";  
        System.out.println("✓ 重复插入测试通过");  
  
        // 测试删除不存在的元素  
        assert !collection.remove(999) : "删除不存在的元素应该返回 false";  
        System.out.println("✓ 删除不存在元素测试通过");  
  
        // 测试删除单个实例  
        assert collection.remove(1) : "删除第一个实例应该成功";  
        assert collection.getRandom() == 1 : "删除一个实例后应该还能获取到值";  
        System.out.println("✓ 删除单个实例测试通过");  
    }  
  
    /**  
     * 测试性能和大数据量  
     */  
    public static void testPerformance() {  
        System.out.println("\n==== 测试性能和大数据量 ====");  
  
        Code04_InsertDeleteRandomDuplicatesAllowed collection = new
```

```
Code04_InsertDeleteRandomDuplicatesAllowed() ;  
    int n = 10000;  
    long startTime = System.currentTimeMillis();  
  
    // 批量插入（包含重复）  
    for (int i = 0; i < n; i++) {  
        collection.insert(i % 100); // 插入 0-99 的重复值  
    }  
  
    // 批量删除  
    for (int i = 0; i < n; i += 2) {  
        collection.remove(i % 100);  
    }  
  
    // 随机操作混合  
    for (int i = 0; i < n; i++) {  
        if (i % 3 == 0) {  
            collection.insert(i % 50 + 100); // 插入 100-149 的新值  
        } else if (i % 5 == 0) {  
            collection.remove(i % 50);  
        } else {  
            collection.getRandom();  
        }  
    }  
  
    long endTime = System.currentTimeMillis();  
    System.out.println("✓ 性能测试通过，处理 " + n + " 次操作耗时：" + (endTime -  
startTime) + "ms");  
}  
  
/**  
 * 测试复杂删除场景  
 */  
public static void testComplexRemoval() {  
    System.out.println("\n==== 测试复杂删除场景 ===");  
  
    Code04_InsertDeleteRandomDuplicatesAllowed collection = new  
Code04_InsertDeleteRandomDuplicatesAllowed();  
  
    // 插入多个重复值  
    collection.insert(1);  
    collection.insert(1);  
    collection.insert(1);
```

```

        collection.insert(2);
        collection.insert(2);
        collection.insert(3);

        // 验证初始状态
        assert collection.remove(1) : "删除第一个 1 应该成功";
        assert collection.getRandom() != 999 : "getRandom 应该返回有效值";

        // 继续删除
        assert collection.remove(1) : "删除第二个 1 应该成功";
        assert collection.remove(1) : "删除第三个 1 应该成功";
        assert !collection.remove(1) : "删除第四个 1 应该失败 (已不存在)";

        // 验证最终状态
        assert collection.remove(2) : "删除第一个 2 应该成功";
        assert collection.remove(2) : "删除第二个 2 应该成功";
        assert collection.remove(3) : "删除 3 应该成功";

        System.out.println("✓ 复杂删除场景测试通过");
    }

    /**
     * 运行所有测试
     */
    public static void runAllTests() {
        try {
            testEdgeCases();
            testPerformance();
            testComplexRemoval();
            System.out.println("\n🎉 所有 RandomizedCollection 测试通过！功能正常。");
        } catch (AssertionError e) {
            System.out.println("✗ 测试失败: " + e.getMessage());
        }
    }

    /**
     * 功能演示
     */
    public static void demonstrate() {
        System.out.println("\n== RandomizedCollection 功能演示 ===");
    }
}

Code04_InsertDeleteRandomDuplicatesAllowed collection = new

```

```
Code04_InsertDeleteRandomDuplicatesAllowed() ;  
  
    System.out.println("1. 插入重复元素 1, 1, 1");  
    collection.insert(1);  
    collection.insert(1);  
    collection.insert(1);  
  
    System.out.println("2. 插入元素 2, 2");  
    collection.insert(2);  
    collection.insert(2);  
  
    System.out.println("3. 删除第一个 1: " + collection.remove(1));  
    System.out.println("4. 删除第二个 1: " + collection.remove(1));  
    System.out.println("5. 删除第三个 1: " + collection.remove(1));  
    System.out.println("6. 尝试删除第四个 1: " + collection.remove(1));  
  
    System.out.println("7. 随机获取元素:");  
    for (int i = 0; i < 5; i++) {  
        System.out.println(" 第" + (i+1) + "次随机: " + collection.getRandom());  
    }  
  
    System.out.println("8. 插入新元素 3, 4");  
    collection.insert(3);  
    collection.insert(4);  
  
    System.out.println("9. 最终随机抽样:");  
    for (int i = 0; i < 3; i++) {  
        System.out.println(" 随机值: " + collection.getRandom());  
    }  
  
    System.out.println("\n演示完成!");  
}  
  
/**  
 * 主函数 - 运行测试和演示  
 */  
public static void main(String[] args) {  
    // 运行单元测试  
    RandomizedCollectionTest.runAllTests();  
  
    // 功能演示  
    demonstrate();  
}
```

}

=====

文件: Code04_InsertDeleteRandomDuplicatesAllowed.py

=====

```
import random
from collections import defaultdict

# 插入、删除和获取随机元素 O(1)时间且允许有重复数字的结构
,,,
```

一、题目解析

设计一个支持在平均时间复杂度 $O(1)$ 下执行以下操作的数据结构（允许重复元素）：

1. insert(val): 将一个元素 val 插入到集合中，返回 true
2. remove(val): 如果元素 val 存在，则从中删除一个实例，返回 true，否则返回 false
3. getRandom: 随机返回集合中的一个元素，每个元素被返回的概率与其在集合中的数量成线性关系

二、算法思路

与不允许重复的版本相比，主要变化在于需要处理重复元素：

1. 使用数组(list)存储所有元素，实现 $O(1)$ 时间复杂度的随机访问
2. 使用字典(defaultdict)存储元素值到其在数组中索引集合的映射
3. 插入操作：在数组末尾添加元素，并在字典中记录其索引
4. 删除操作：将要删除的元素与数组末尾元素交换，然后删除末尾元素，更新字典
5. 随机获取：使用 random 模块随机生成索引，访问数组中对应元素

三、时间复杂度分析

insert 操作: $O(1)$ - 数组末尾插入 + 字典更新

remove 操作: $O(1)$ - 字典查找 + 数组元素交换 + 数组末尾删除 + 字典更新

getRandom 操作: $O(1)$ - 随机索引生成 + 数组访问

四、空间复杂度分析

$O(n)$ - n 为集合中元素个数，需要数组和字典分别存储元素和索引映射

五、工程化考量

1. 异常处理：处理空集合的 getRandom 操作
2. 边界场景：空集合、单元素集合等
3. 随机性：确保 getRandom 方法能真正按概率返回每个元素

六、相关题目扩展

1. LeetCode 381. 常数时间插入、删除和获取随机元素-允许重复（本题）
2. LeetCode 380. 常数时间插入、删除和获取随机元素
3. 牛客网相关题目

,,

```
class RandomizedCollection:

    def __init__(self):
        """构造函数"""
        # 字典存储元素值到其在数组中索引集合的映射
        self.map = defaultdict(set)
        # 数组存储所有元素值
        self.arr = []

    def insert(self, val: int) -> bool:
        """
        插入元素
        :param val: 要插入的元素
        :return: 总是返回 true
        时间复杂度: O(1)
        """
        # 在数组末尾添加元素
        self.arr.append(val)
        # 将新索引添加到该元素值对应的索引集合中
        self.map[val].add(len(self.arr) - 1)
        # 当且仅当该元素第一次插入时返回 true
        return len(self.map[val]) == 1

    def remove(self, val: int) -> bool:
        """
        删除元素
        :param val: 要删除的元素
        :return: 如果元素存在则删除并返回 true, 否则返回 false
        时间复杂度: O(1)
        """
        # 检查元素是否存在
        if val not in self.map:
            return False
        # 获取该元素值对应的索引集合
        val_set = self.map[val]
        # 获取其中一个索引 (任意一个)
        val_any_index = next(iter(val_set))
        # 获取数组末尾元素的值
        end_value = self.arr[-1]
        # 如果要删除的元素就是末尾元素
        if val == end_value:
            # 直接从索引集合中删除该索引
            val_set.discard(len(self.arr) - 1)
```

```
else:
    # 获取末尾元素值对应的索引集合
    end_value_set = self.map[end_value]
    # 将末尾元素的索引更新为要删除元素的索引
    end_value_set.add(val_any_index)
    # 更新数组中要删除元素位置的值为末尾元素值
    self.arr[val_any_index] = end_value
    # 从末尾元素的索引集合中删除原末尾索引
    end_value_set.discard(len(self.arr) - 1)
    # 从要删除元素的索引集合中删除该索引
    val_set.discard(val_any_index)
    # 删除数组末尾元素
    self.arr.pop()
    # 如果要删除元素的索引集合为空，则从字典中删除该元素
    if not val_set:
        del self.map[val]
return True

def getRandom(self) -> int:
    """
随机获取元素
:return: 随机返回集合中的一个元素
时间复杂度: O(1)
"""
    # 检查集合是否为空
    if not self.arr:
        raise Exception("集合为空，无法获取随机元素")
    # 随机返回数组中的一个元素
    return random.choice(self.arr)

# 测试代码
if __name__ == "__main__":
    collection = RandomizedCollection()

    # 简单测试
    print("insert(1):", collection.insert(1)) # True
    print("insert(1):", collection.insert(1)) # False
    print("insert(2):", collection.insert(2)) # True
    print("remove(1):", collection.remove(1)) # True
    print("getRandom:", collection.getRandom()) # 1 or 2
=====
```

文件: Code05_MedianFinder.cpp

```
=====
```

```
#include <queue>
#include <vector>
#include <functional>
#include <stdexcept>
#include <iostream>

// 快速获得数据流的中位数的结构
/*
 * 一、题目解析
 * 设计一个支持以下两种操作的数据结构：
 * 1. void addNum(int num) - 从数据流中添加一个整数到数据结构中
 * 2. double findMedian() - 返回目前所有元素的中位数
 *
 * 中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。
 *
 * 二、算法思路
 * 使用两个优先队列（堆）来维护数据：
 * 1. maxHeap (最大堆): 存储较小的一半元素
 * 2. minHeap (最小堆): 存储较大的一半元素
 *
 * 保持两个堆的大小平衡：
 * 1. 当元素总数为偶数时，两个堆大小相等
 * 2. 当元素总数为奇数时，maxHeap 比 minHeap 多一个元素
 *
 * 三、时间复杂度分析
 * addNum 操作: O(log n) - 堆的插入和调整操作
 * findMedian 操作: O(1) - 直接访问堆顶元素
 *
 * 四、空间复杂度分析
 * O(n) - n 为添加的元素个数，需要两个堆分别存储元素
 *
 * 五、工程化考量
 * 1. 异常处理：处理空数据流的 findMedian 操作
 * 2. 边界场景：空数据流、单元素数据流等
 * 3. 数值精度：注意整数除法的精度问题
 * 4. 内存管理：C++需要手动管理内存
 *
 * 六、相关题目扩展
 * 1. LeetCode 295. 数据流的中位数 (本题)
 * 2. LeetCode 480. 滑动窗口中位数
 * 3. 牛客网：数据流中的中位数
```

* 4. 剑指 Offer 41. 数据流中的中位数

*/

```
class MedianFinder {  
private:  
    // 最大堆，存储较小的一半元素  
    std::priority_queue<int> maxHeap;  
  
    // 最小堆，存储较大的一半元素  
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;  
  
public:  
    // 构造函数  
    MedianFinder() = default;  
  
    /*  
     * 添加数字到数据结构中  
     * @param num 要添加的数字  
     * 时间复杂度: O(log n)  
     */  
    void addNum(int num) {  
        // 如果最大堆为空或新数字小于等于最大堆堆顶，则添加到最大堆  
        if (maxHeap.empty() || maxHeap.top() >= num) {  
            maxHeap.push(num);  
        } else {  
            // 否则添加到最小堆  
            minHeap.push(num);  
        }  
        // 平衡两个堆的大小  
        balance();  
    }  
  
    /*  
     * 查找当前所有元素的中位数  
     * @return 中位数  
     * 时间复杂度: O(1)  
     */  
    double findMedian() {  
        // 检查数据流是否为空  
        if (maxHeap.empty() && minHeap.empty()) {  
            throw std::runtime_error("数据流为空，无法获取中位数");  
        }  
        // 如果两个堆大小相等，返回两个堆堆顶的平均值  
    }
```

```

    if (maxHeap.size() == minHeap.size()) {
        return static_cast<double>(maxHeap.top() + minHeap.top()) / 2;
    } else {
        // 否则返回较大堆的堆顶元素
        return maxHeap.size() > minHeap.size() ? maxHeap.top() : minHeap.top();
    }
}

private:
/*
 * 平衡两个堆的大小
 * 确保两个堆的大小差不超过 1
 */
void balance() {
    // 如果最大堆比最小堆多超过 1 个元素，则移动一个元素到最小堆
    if (maxHeap.size() > minHeap.size() + 1) {
        minHeap.push(maxHeap.top());
        maxHeap.pop();
    }
    // 如果最小堆比最大堆多超过 1 个元素，则移动一个元素到最大堆
    else if (minHeap.size() > maxHeap.size() + 1) {
        maxHeap.push(minHeap.top());
        minHeap.pop();
    }
}
};

// 测试代码
int main() {
    MedianFinder medianFinder;

    // 测试用例: ["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
    //           [[], [1], [2], [], [3], []]

    medianFinder.addNum(1);      // arr = [1]
    medianFinder.addNum(2);      // arr = [1, 2]
    std::cout << "中位数: " << medianFinder.findMedian() << std::endl; // 返回 1.5 ((1 + 2) / 2)
    medianFinder.addNum(3);      // arr[1, 2, 3]
    std::cout << "中位数: " << medianFinder.findMedian() << std::endl; // 返回 2.0

    return 0;
}

```

文件: Code05_MedianFinder.java

```
=====
package class035;
```

```
import java.util.*;
```

```
/**
 * 实时中位数查找器 - 支持动态数据流的中位数查找
 * 题目来源: LeetCode 295. Find Median from Data Stream
 * 网址: https://leetcode.com/problems/find-median-from-data-stream/
 *
 * 一、题目解析
 * 设计一个支持以下操作的数据结构:
 * 1. void addNum(int num): 从数据流中添加一个整数
 * 2. double findMedian(): 返回目前所有元素的中位数
 *
 * 中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。
 *
 * 二、算法思路
 * 使用两个堆来维护数据流的中位数:
 * 1. 最大堆(maxHeap): 存储较小的一半元素，堆顶是最大值
 * 2. 最小堆(minHeap): 存储较大的一半元素，堆顶是最小值
 * 3. 保持两个堆的大小平衡: 最大堆的大小  $\geq$  最小堆的大小，且差值不超过 1
 * 4. 保证最大堆的所有元素  $\leq$  最小堆的所有元素
 *
 * 三、时间复杂度分析
 * - addNum():  $O(\log n)$  堆插入操作
 * - findMedian():  $O(1)$  直接访问堆顶元素
 *
 * 四、空间复杂度分析
 *  $O(n)$ , 需要存储所有元素
 *
 * 五、工程化考量
 * 1. 异常处理: 处理空数据流的 findMedian 操作
 * 2. 边界情况: 空数据流、单个元素、两个元素等特殊情况
 * 3. 内存管理: Java 自动垃圾回收，但仍需注意大对象的内存消耗
 * 4. 线程安全: 当前实现非线程安全，如需线程安全可使用同步机制
 * 5. 性能优化: 利用堆的特性实现  $O(\log n)$  插入和  $O(1)$  查询
 * 6. 可扩展性: 可扩展为支持泛型或更多统计功能
 *
 * 六、相关题目扩展
```

- * 1. LeetCode 295. [Find Median from Data Stream] (<https://leetcode.com/problems/find-median-from-data-stream/>) (本题)
- * 2. LeetCode 480. [Sliding Window Median] (<https://leetcode.com/problems/sliding-window-median/>) (滑动窗口中位数)
- * 3. 牛客网: [数据流中的中位数] (<https://www.nowcoder.com/practice/9be0172896bd43948f8a32fb954e1be1>)
- * 4. 剑指 Offer 41. [数据流中的中位数] (<https://leetcode.cn/problems/shu-ju-liu-zhong-de-zhong-wei-shu-lcof/>)
- * 5. LintCode 81. [Find Median from Data Stream] (<https://www.lintcode.com/problem/81/>)
- * 6. HackerRank: [Heaps - Find the Running Median] (<https://www.hackerrank.com/challenges/ctci-find-the-running-median/problem>)
- * 7. CodeChef: [Median of Medians] (<https://www.codechef.com/problems/MEDIAN>)
- * 8. 计蒜客: [数据流中位数] (<https://nanti.jisuanke.com/t/41395>)
- *
- * 七、补充题目（各大 OJ 平台）
 - * 1. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) – 数据流处理优化
 - * 2. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) – 消息流处理
 - * 3. UVA 11525.
[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2520) – 排列中位数问题
 - * 4. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) – 区间中位数查询
 - * 5. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) – 中位数计数优化
 - * 6. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) – 动态规划中位数优化
 - * 7. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) – 数据流唯一性检测
 - * 8. USACO Training: [Median Finder] (<https://train.usaco.org/>) – 中位数基础训练
 - * 9. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) – 本题的简化版本
 - * 10. 赛码: [数据流处理] (<https://www.acmcoder.com/>) – 在线编程题目
 - *
- * 八、算法设计技巧总结
 - * 1. 双堆平衡: 使用两个堆维护数据的有序性, 通过平衡机制保证中位数的快速获取
 - * 2. 分治思想: 将数据分为两部分, 较小的一半和较大的一半
 - * 3. 堆特性利用: 最大堆维护较小一半的最大值, 最小堆维护较大一半的最小值
 - * 4. 动态平衡: 每次插入后重新平衡两个堆的大小
 - * 5. 边界处理: 特殊处理堆大小相等和不等的情况
 - *
- * 九、面试要点
 - * 1. 解释为什么使用两个堆而不是一个排序数组
 - * 2. 分析堆平衡策略的必要性和实现方式
 - * 3. 讨论各种边界情况的处理

- * 4. 分析时间复杂度和空间复杂度
- * 5. 提出可能的扩展和优化方向
- * 6. 讨论线程安全性问题和解决方案

*

* 十、工程实践中的应用场景

- * 1. 实时数据分析系统中的统计计算
- * 2. 金融系统中的价格中位数计算
- * 3. 网络监控系统中的延迟分析
- * 4. 推荐系统中的评分中位数计算
- * 5. 数据库查询优化中的统计信息维护
- * 6. 游戏开发中的排行榜中位数计算

*/

```
public class Code05_MedianFinder {
```

```
    private PriorityQueue<Integer> maxHeap; // 存储较小的一半（最大堆）  
    private PriorityQueue<Integer> minHeap; // 存储较大的一半（最小堆）
```

```
/** 初始化数据结构 */
```

```
public Code05_MedianFinder() {  
    // 最大堆：存储较小的一半，堆顶是最大值  
    maxHeap = new PriorityQueue<>(Collections.reverseOrder());  
    // 最小堆：存储较大的一半，堆顶是最小值  
    minHeap = new PriorityQueue<>();  
}
```

```
/**
```

```
 * 添加数字到数据流  
 * @param num 要添加的数字  
 * 时间复杂度：O(log n) 堆插入操作  
 * 空间复杂度：O(1)  
 * 核心思想：维护两个堆的平衡和元素关系  
 */
```

```
public void addNum(int num) {  
    // 先加入最大堆（较小的一半）  
    maxHeap.offer(num);
```

```
    // 平衡两个堆，确保最大堆的所有元素 <= 最小堆的所有元素  
    // 将最大堆的最大值移到最小堆  
    minHeap.offer(maxHeap.poll());
```

```
    // 保持两个堆的大小平衡（最大堆大小 >= 最小堆大小）  
    // 如果最小堆比最大堆大，则将最小堆的最小值移到最大堆  
    if (maxHeap.size() < minHeap.size()) {
```

```

        maxHeap.offer(minHeap.poll());
    }
}

/***
 * 查找当前数据流的中位数
 * @return 中位数
 * 时间复杂度: O(1) 直接访问堆顶元素
 * 空间复杂度: O(1)
 * 核心思想: 根据两个堆的大小关系计算中位数
 */
public double findMedian() {
    if (maxHeap.size() == minHeap.size()) {
        // 偶数个元素, 取两个堆顶的平均值
        return (maxHeap.peek() + minHeap.peek()) / 2.0;
    } else {
        // 奇数个元素, 取最大堆的堆顶 (因为最大堆多一个元素)
        return maxHeap.peek();
    }
}

// ===== 单元测试和功能演示 =====

/***
 * 单元测试类 - 测试 MedianFinder 的各种功能
 */
public static class MedianFinderTest {

    /**
     * 测试边界情况
     */
    public static void testEdgeCases() {
        System.out.println("\n==== 测试边界情况 ===");

        Code05_MedianFinder finder = new Code05_MedianFinder();

        // 测试空数据流
        try {
            finder.findMedian();
            assertFalse("空数据流应该抛出异常");
        } catch (Exception e) {
            System.out.println("✓ 空数据流异常处理正确");
        }
    }
}

```

```
// 测试单个元素
finder.addNum(5);
assert finder.findMedian() == 5.0 : "单个元素中位数应该是 5.0";
System.out.println("✓ 单个元素测试通过");

// 测试两个元素
finder.addNum(10);
assert finder.findMedian() == 7.5 : "两个元素中位数应该是 7.5";
System.out.println("✓ 两个元素测试通过");

// 测试三个元素
finder.addNum(2);
assert finder.findMedian() == 5.0 : "三个元素中位数应该是 5.0";
System.out.println("✓ 三个元素测试通过");
}

/**
 * 测试性能和大数据量
 */
public static void testPerformance() {
    System.out.println("\n==== 测试性能和大数据量 ====");

    Code05_MedianFinder finder = new Code05_MedianFinder();
    int n = 10000;
    long startTime = System.currentTimeMillis();

    // 批量添加数字
    Random random = new Random();
    for (int i = 0; i < n; i++) {
        finder.addNum(random.nextInt(1000));
        if (i % 1000 == 0) {
            finder.findMedian(); // 每隔 1000 次查询一次中位数
        }
    }

    long endTime = System.currentTimeMillis();
    System.out.println("✓ 性能测试通过, 处理 " + n + " 个元素耗时: " + (endTime - startTime) + "ms");

    // 验证最终中位数合理性
    double median = finder.findMedian();
    assert median >= 0 && median <= 1000 : "中位数应该在合理范围内";
}
```

```

        System.out.println("✓ 最终中位数: " + median);
    }

    /**
     * 测试随机数据
     */
    public static void testRandomData() {
        System.out.println("\n==== 测试随机数据 ====");

        Code05_MedianFinder finder = new Code05_MedianFinder();
        Random random = new Random();
        List<Integer> numbers = new ArrayList<>();

        // 添加随机数据
        for (int i = 0; i < 100; i++) {
            int num = random.nextInt(100);
            finder.addNum(num);
            numbers.add(num);

            // 验证中位数计算正确性
            Collections.sort(numbers);
            double expectedMedian;
            if (numbers.size() % 2 == 0) {
                expectedMedian = (numbers.get(numbers.size()/2 - 1) +
numbers.get(numbers.size()/2)) / 2.0;
            } else {
                expectedMedian = numbers.get(numbers.size()/2);
            }

            double actualMedian = finder.findMedian();
            assert Math.abs(actualMedian - expectedMedian) < 0.0001 :
                "中位数计算错误: 期望 " + expectedMedian + ", 实际 " + actualMedian;
        }

        System.out.println("✓ 随机数据中位数计算正确");
    }

    /**
     * 测试堆平衡机制
     */
    public static void testHeapBalance() {
        System.out.println("\n==== 测试堆平衡机制 ====");

        Code05_MedianFinder finder = new Code05_MedianFinder();

```

```
// 添加递增序列
for (int i = 1; i <= 10; i++) {
    finder.addNum(i);
}

// 验证堆大小平衡
// 由于我们的实现，最大堆应该比最小堆多 0 或 1 个元素
int sizeDiff = Math.abs(finder.maxHeap.size() - finder.minHeap.size());
assert sizeDiff <= 1 : "堆大小不平衡，差值：" + sizeDiff;
System.out.println("✓ 堆大小平衡测试通过");

// 验证堆顶元素关系
if (!finder.maxHeap.isEmpty() && !finder.minHeap.isEmpty()) {
    assert finder.maxHeap.peek() <= finder.minHeap.peek() :
        "最大堆顶应该 <= 最小堆顶";
    System.out.println("✓ 堆顶元素关系正确");
}

// 验证中位数计算
double median = finder.findMedian();
assert median == 5.5 : "1-10 序列中位数应该是 5.5，实际：" + median;
System.out.println("✓ 中位数计算正确");
}

/**
 * 运行所有测试
 */
public static void runAllTests() {
    try {
        testEdgeCases();
        testPerformance();
        testRandomData();
        testHeapBalance();
        System.out.println("\n🎉 所有 MedianFinder 测试通过！功能正常。");
    } catch (AssertionError e) {
        System.out.println("✖ 测试失败：" + e.getMessage());
    }
}

/**
 * 功能演示
```

```
/*
public static void demonstrate() {
    System.out.println("\n==== MedianFinder 功能演示 ===");

    Code05_MedianFinder finder = new Code05_MedianFinder();

    System.out.println("1. 添加数字: 1, 2, 3");
    finder.addNum(1);
    finder.addNum(2);
    finder.addNum(3);
    System.out.println("    当前中位数: " + finder.findMedian());

    System.out.println("2. 添加数字: 4");
    finder.addNum(4);
    System.out.println("    当前中位数: " + finder.findMedian());

    System.out.println("3. 添加数字: 5");
    finder.addNum(5);
    System.out.println("    当前中位数: " + finder.findMedian());

    System.out.println("4. 添加数字: 0");
    finder.addNum(0);
    System.out.println("    当前中位数: " + finder.findMedian());

    System.out.println("5. 添加数字: 6");
    finder.addNum(6);
    System.out.println("    当前中位数: " + finder.findMedian());

    System.out.println("\n演示完成!");
}

/**
 * 主函数 - 运行测试和演示
 */
public static void main(String[] args) {
    // 运行单元测试
    MedianFinderTest.runAllTests();

    // 功能演示
    demonstrate();
}
```

文件: Code05_MedianFinder.py

```
=====import heapq
```

```
# 快速获得数据流的中位数的结构
```

```
,,
```

一、题目解析

设计一个支持以下两种操作的数据结构:

1. void addNum(int num) - 从数据流中添加一个整数到数据结构中
2. double findMedian() - 返回目前所有元素的中位数

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

二、算法思路

使用两个堆来维护数据:

1. max_heap (最大堆): 存储较小的一半元素，使用负数模拟最大堆
2. min_heap (最小堆): 存储较大的一半元素

保持两个堆的大小平衡:

1. 当元素总数为偶数时，两个堆大小相等
2. 当元素总数为奇数时，max_heap 比 min_heap 多一个元素

三、时间复杂度分析

addNum 操作: $O(\log n)$ - 堆的插入和调整操作

findMedian 操作: $O(1)$ - 直接访问堆顶元素

四、空间复杂度分析

$O(n)$ - n 为添加的元素个数，需要两个堆分别存储元素

五、工程化考量

1. 异常处理: 处理空数据流的 findMedian 操作
2. 边界场景: 空数据流、单元素数据流等
3. 数值精度: 注意整数除法的精度问题

六、相关题目扩展

1. LeetCode 295. 数据流的中位数 (本题)
 2. LeetCode 480. 滑动窗口中位数
 3. 牛客网: 数据流中的中位数
 4. 剑指 Offer 41. 数据流中的中位数
- ,,

```
class MedianFinder:
    def __init__(self):
        """构造函数"""
        # Python 的 heapq 是最小堆，使用负数来模拟最大堆
        self.max_heap = [] # 存储较小的一半元素（使用负数）
        self.min_heap = [] # 存储较大的一半元素

    def addNum(self, num: int) -> None:
        """
        添加数字到数据结构中
        :param num: 要添加的数字
        时间复杂度: O(log n)
        """
        # 如果最大堆为空或新数字小于等于最大堆堆顶，则添加到最大堆
        if not self.max_heap or -self.max_heap[0] >= num:
            heapq.heappush(self.max_heap, -num)
        else:
            # 否则添加到最小堆
            heapq.heappush(self.min_heap, num)
        # 平衡两个堆的大小
        self._balance()

    def findMedian(self) -> float:
        """
        查找当前所有元素的中位数
        :return: 中位数
        时间复杂度: O(1)
        """
        # 检查数据流是否为空
        if not self.max_heap and not self.min_heap:
            raise Exception("数据流为空，无法获取中位数")
        # 如果两个堆大小相等，返回两个堆堆顶的平均值
        if len(self.max_heap) == len(self.min_heap):
            return (-self.max_heap[0] + self.min_heap[0]) / 2
        else:
            # 否则返回较大堆的堆顶元素
            return -self.max_heap[0] if len(self.max_heap) > len(self.min_heap) else
self.min_heap[0]

    def _balance(self) -> None:
        """
        平衡两个堆的大小
        确保两个堆的大小差不超过 1
        """
```

```

"""
# 如果最大堆比最小堆多超过 1 个元素，则移动一个元素到最小堆
if len(self.max_heap) > len(self.min_heap) + 1:
    heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))
# 如果最小堆比最大堆多超过 1 个元素，则移动一个元素到最大堆
elif len(self.min_heap) > len(self.max_heap) + 1:
    heapq.heappush(self.max_heap, -heapq.heappop(self.min_heap))

# 测试代码
if __name__ == "__main__":
    medianFinder = MedianFinder()

# 测试用例: ["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
#           [[], [1], [2], [], [3], []]

medianFinder.addNum(1)      # arr = [1]
medianFinder.addNum(2)      # arr = [1, 2]
print("中位数:", medianFinder.findMedian())  # 返回 1.5 ((1 + 2) / 2)
medianFinder.addNum(3)      # arr[1, 2, 3]
print("中位数:", medianFinder.findMedian())  # 返回 2.0

```

文件: Code06_MaximumFrequencyStack.cpp

```

#include <unordered_map>
#include <vector>
#include <stdexcept>
#include <iostream>

// 最大频率栈
/*
 * 一、题目解析
 * 实现一个类似栈的数据结构，支持以下操作：
 * 1. push(val)：将一个整数 val 压入栈顶
 * 2. pop()：删除并返回栈中出现频率最高的元素
 *   如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素
 *
 * 二、算法思路
 * 使用两个哈希表来维护数据：
 * 1. valueTimes：记录每个值的出现频率
 * 2. cntValues：记录每个频率对应的值列表（使用 vector 实现）
 * 3. topTimes：记录当前最大频率

```

```
*  
* push 操作:  
* 1. 更新值的频率  
* 2. 将值添加到对应频率的列表中  
* 3. 更新最大频率  
*  
* pop 操作:  
* 1. 从最大频率对应的列表中移除最后一个元素  
* 2. 更新该元素的频率  
* 3. 如果最大频率列表为空，则减少最大频率  
*  
* 三、时间复杂度分析  
* push 操作: O(1) - 哈希表操作和列表操作都是 O(1)  
* pop 操作: O(1) - 哈希表操作和列表操作都是 O(1)  
*  
* 四、空间复杂度分析  
* O(n) - n 为 push 操作的次数，需要存储所有元素及其频率信息  
*  
* 五、工程化考量  
* 1. 异常处理：处理空栈的 pop 操作  
* 2. 边界场景：空栈、单元素栈等  
* 3. 内存管理：C++ 需要手动管理内存  
*  
* 六、相关题目扩展  
* 1. LeetCode 895. 最大频率栈（本题）  
* 2. 牛客网：最大频率栈  
* 3. 剑指 Offer 相关栈题目  
*/
```

```
class FreqStack {  
private:  
    // 出现的最大次数  
    int topTimes;  
    // 每层节点，频率到值列表的映射  
    std::unordered_map<int, std::vector<int>> cntValues;  
    // 每一个数出现了几次，值到频率的映射  
    std::unordered_map<int, int> valueTimes;  
  
public:  
    // 构造函数  
    FreqStack() : topTimes(0) {}  
  
    /*
```

```

* 压入元素到栈中
* @param val 要压入的元素
* 时间复杂度: O(1)
*/
void push(int val) {
    // 更新值的频率
    valueTimes[val] = valueTimes.count(val) ? valueTimes[val] + 1 : 1;
    int curTopTimes = valueTimes[val];
    // 将值添加到对应频率的列表中
    cntValues[curTopTimes].push_back(val);
    // 更新最大频率
    topTimes = std::max(topTimes, curTopTimes);
}

/*
* 弹出频率最高的元素
* @return 频率最高的元素, 如果有多个则返回最接近栈顶的
* 时间复杂度: O(1)
*/
int pop() {
    // 检查栈是否为空
    if (topTimes == 0) {
        throw std::runtime_error("栈为空, 无法执行 pop 操作");
    }
    // 从最大频率对应的列表中移除最后一个元素
    std::vector<int>& topTimeValues = cntValues[topTimes];
    int ans = topTimeValues.back();
    topTimeValues.pop_back();
    // 如果最大频率列表为空, 则减少最大频率
    if (topTimeValues.empty()) {
        cntValues.erase(topTimes--);
    }
    // 更新弹出元素的频率
    int times = valueTimes[ans];
    if (times == 1) {
        valueTimes.erase(ans);
    } else {
        valueTimes[ans] = times - 1;
    }
    return ans;
}

```

```

// 测试代码
int main() {
    FreqStack freqStack;

    // 测试用例: ["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"]
    //           [[], [5], [7], [5], [7], [4], [5], [], [], []]

    freqStack.push(5); // 堆栈为 [5]
    freqStack.push(7); // 堆栈是 [5, 7]
    freqStack.push(5); // 堆栈是 [5, 7, 5]
    freqStack.push(7); // 堆栈是 [5, 7, 5, 7]
    freqStack.push(4); // 堆栈是 [5, 7, 5, 7, 4]
    freqStack.push(5); // 堆栈是 [5, 7, 5, 7, 4, 5]

    std::cout << "pop(): " << freqStack.pop() << std::endl; // 返回 5
    std::cout << "pop(): " << freqStack.pop() << std::endl; // 返回 7
    std::cout << "pop(): " << freqStack.pop() << std::endl; // 返回 5
    std::cout << "pop(): " << freqStack.pop() << std::endl; // 返回 4

    return 0;
}

```

=====

文件: Code06_MaximumFrequencyStack.java

=====

```

package class035;

import java.util.*;

/**
 * 最大频率栈 - 支持按频率弹出元素的栈结构
 * 题目来源: LeetCode 895. Maximum Frequency Stack
 * 网址: https://leetcode.com/problems/maximum-frequency-stack/
 *
 * 一、题目解析
 * 实现一个类似栈的数据结构，支持以下操作：
 * 1. void push(int val): 将整数 val 压入栈中
 * 2. int pop(): 删除并返回出现频率最高的元素；如果有多个元素频率相同，
 *   则删除并返回最近压入的那个元素
 *
 * 二、算法思路
 * 使用三个数据结构来实现最大频率栈：

```

- * 1. frequency Map<Integer, Integer>: 记录每个元素的出现频率
- * 2. group Map<Integer, Stack<Integer>>: 按频率分组存储元素，键为频率，值为该频率下元素的栈
- * 3. maxFrequency int: 记录当前最大频率
- *
- * 核心思想:
 - * - push 操作时更新元素频率，并将元素添加到对应频率的栈中
 - * - pop 操作时从最大频率的栈中弹出元素，并更新相关频率信息
- *
- * 三、时间复杂度分析
 - * - push(): O(1) 平均时间复杂度
 - * - pop(): O(1) 平均时间复杂度
- *
- * 四、空间复杂度分析
 - * O(n)，需要存储所有元素及其频率信息
- *
- * 五、工程化考量
 - * 1. 异常处理：处理空栈的 pop 操作
 - * 2. 边界情况：空栈、单个元素、重复元素等特殊情况
 - * 3. 内存管理：Java 自动垃圾回收，但仍需注意大对象的内存消耗
 - * 4. 线程安全：当前实现非线程安全，如需线程安全可使用同步机制
 - * 5. 性能优化：利用哈希表和栈的特性实现 O(1) 操作
 - * 6. 可扩展性：可扩展为支持泛型或更多统计功能
- *
- * 六、相关题目扩展
 - * 1. LeetCode 895. [Maximum Frequency Stack] (<https://leetcode.com/problems/maximum-frequency-stack/>) (本题)
 - * 2. LeetCode 146. [LRU Cache] (<https://leetcode.com/problems/lru-cache/>) (LRU 缓存)
 - * 3. 牛客网：[设计最大频率栈] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>)
 - * 4. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
 - * 5. LintCode 1286. [最小操作数] (<https://www.lintcode.com/problem/1286/>)
 - * 6. HackerRank: [Stacks – Maximum Element] (<https://www.hackerrank.com/challenges/maximum-element/problem>)
 - * 7. CodeChef: [Frequency Stack] (<https://www.codechef.com/problems/FREQSTK>)
 - * 8. 计蒜客：[频率栈设计] (<https://nanti.jisuanke.com/t/41396>)
- *
- * 七、补充题目（各大 OJ 平台）
 - * 1. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) – 频率统计优化
 - * 2. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) – 消息频率处理
 - * 3. UVA 11525. [Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem)

&problem=2520) - 排列频率问题

- * 4. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) - 区间频率查询
- * 5. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) - 频率计数优化
- * 6. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) - 动态规划频率优化
- * 7. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) - 唯一性频率检测
- * 8. USACO Training: [Frequency Stack] (<https://train.usaco.org/>) - 频率栈基础训练
- * 9. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 数据流频率
- * 10. 赛码: [频率数据结构] (<https://www.acmcoder.com/>) - 在线编程题目

*

* 八、算法设计技巧总结

- * 1. 频率分组: 按频率将元素分组存储, 便于快速查找最大频率元素
- * 2. 栈结构维护: 利用栈的后进先出特性处理相同频率元素的弹出顺序
- * 3. 动态频率更新: 每次 push/pop 操作都动态更新元素频率和最大频率
- * 4. 空间换时间: 使用额外的哈希表和栈结构实现 O(1) 操作
- * 5. 边界处理: 特殊处理频率变化和栈空情况

*

* 九、面试要点

- * 1. 解释为什么需要按频率分组存储元素
- * 2. 分析最大频率更新策略的必要性和实现方式
- * 3. 讨论各种边界情况的处理
- * 4. 分析时间复杂度和空间复杂度
- * 5. 提出可能的扩展和优化方向
- * 6. 讨论线程安全性问题和解决方案

*

* 十、工程实践中的应用场景

- * 1. 缓存系统中的热点数据管理
- * 2. 推荐系统中的热门内容排序
- * 3. 网络监控系统中的高频事件检测
- * 4. 日志分析系统中的高频错误统计
- * 5. 游戏开发中的热门道具管理
- * 6. 数据库查询优化中的高频查询统计

*/

```
public class Code06_MaximumFrequencyStack {  
  
    private Map<Integer, Integer> frequency;          // 元素到频率的映射  
    private Map<Integer, Stack<Integer>> group;        // 频率到元素栈的映射  
    private int maxFrequency;                          // 当前最大频率  
  
    /** 初始化数据结构 */  
    public Code06_MaximumFrequencyStack() {  
        frequency = new HashMap<>();  
    }
```

```
group = new HashMap<>();
maxFrequency = 0;
}

/**
 * 压入元素到栈中
 * @param val 要压入的元素
 * 时间复杂度: O(1) 平均时间复杂度
 * 空间复杂度: O(1)
 * 核心思想: 更新元素频率, 将元素添加到对应频率的栈中, 并更新最大频率
 */
public void push(int val) {
    // 更新频率
    int freq = frequency.getOrDefault(val, 0) + 1;
    frequency.put(val, freq);

    // 更新最大频率
    if (freq > maxFrequency) {
        maxFrequency = freq;
    }

    // 将元素添加到对应频率的栈中
    group.computeIfAbsent(freq, k -> new Stack<>()).push(val);
}

/**
 * 弹出频率最高的元素 (如果多个元素频率相同, 弹出最近压入的)
 * @return 弹出的元素
 * 时间复杂度: O(1) 平均时间复杂度
 * 空间复杂度: O(1)
 * 核心思想: 从最大频率的栈中弹出元素, 并更新相关频率信息
 */
public int pop() {
    // 获取最大频率对应的栈
    Stack<Integer> stack = group.get(maxFrequency);
    int val = stack.pop();

    // 更新频率
    frequency.put(val, frequency.get(val) - 1);

    // 如果当前最大频率的栈为空, 减少最大频率
    if (stack.isEmpty()) {
        maxFrequency--;
    }
}
```

```
}
```

```
    return val;
```

```
}
```

```
// ===== 单元测试和功能演示 =====
```

```
/**
```

```
* 单元测试类 - 测试 FreqStack 的各种功能
```

```
*/
```

```
public static class FreqStackTest {
```

```
/**
```

```
* 测试边界情况
```

```
*/
```

```
public static void testEdgeCases() {
```

```
    System.out.println("\n==== 测试边界情况 ===");
```

```
    Code06_MaximumFrequencyStack stack = new Code06_MaximumFrequencyStack();
```

```
    // 测试空栈弹出
```

```
    try {
```

```
        stack.pop();
```

```
        assert false : "空栈弹出应该抛出异常";
```

```
    } catch (Exception e) {
```

```
        System.out.println("✓ 空栈异常处理正确");
```

```
}
```

```
    // 测试单个元素
```

```
    stack.push(1);
```

```
    assert stack.pop() == 1 : "单个元素弹出应该是 1";
```

```
    System.out.println("✓ 单个元素测试通过");
```

```
    // 测试重复压入弹出
```

```
    stack.push(2);
```

```
    stack.push(2);
```

```
    stack.push(3);
```

```
    // 验证弹出顺序 (频率高的先出)
```

```
    assert stack.pop() == 2 : "第一次弹出应该是频率最高的 2";
```

```
    assert stack.pop() == 2 : "第二次弹出应该是另一个 2";
```

```
    assert stack.pop() == 3 : "第三次弹出应该是 3";
```

```
    System.out.println("✓ 重复元素测试通过");
```

```
}

/**
 * 测试性能和大数据量
 */
public static void testPerformance() {
    System.out.println("\n==== 测试性能和大数据量 ===");

    Code06_MaximumFrequencyStack stack = new Code06_MaximumFrequencyStack();
    int n = 10000;
    long startTime = System.currentTimeMillis();

    // 批量压入
    Random random = new Random();
    for (int i = 0; i < n; i++) {
        stack.push(random.nextInt(100)); // 压入 0-99 的随机数
    }

    // 批量弹出
    for (int i = 0; i < n; i++) {
        stack.pop();
    }

    long endTime = System.currentTimeMillis();
    System.out.println("✓ 性能测试通过，处理 " + n + " 次操作耗时：" + (endTime - startTime) + "ms");
}

/**
 * 测试复杂场景
 */
public static void testComplexScenarios() {
    System.out.println("\n==== 测试复杂场景 ===");

    Code06_MaximumFrequencyStack stack = new Code06_MaximumFrequencyStack();

    // 场景 1：多个元素频率相同
    stack.push(1);
    stack.push(2);
    stack.push(1);
    stack.push(2);
    stack.push(3);
```

```
// 验证弹出顺序（频率相同，后进先出）
assert stack.pop() == 2 : "第一次弹出应该是 2 (频率 2)";
assert stack.pop() == 1 : "第二次弹出应该是 1 (频率 2)";
assert stack.pop() == 3 : "第三次弹出应该是 3 (频率 1)";

// 场景 2：频率变化
stack.push(4);
stack.push(4);
stack.push(4);
stack.push(5);
stack.push(5);

assert stack.pop() == 4 : "第一次弹出应该是 4 (频率 3)";
assert stack.pop() == 4 : "第二次弹出应该是 4 (频率 2)";
assert stack.pop() == 5 : "第三次弹出应该是 5 (频率 2)";
assert stack.pop() == 5 : "第四次弹出应该是 5 (频率 1)";
assert stack.pop() == 4 : "第五次弹出应该是 4 (频率 1)";

System.out.println("✓ 复杂场景测试通过");
}
```

```
/***
 * 测试频率跟踪准确性
 */
public static void testFrequencyTracking() {
    System.out.println("\n==== 测试频率跟踪准确性 ===");

    Code06_MaximumFrequencyStack stack = new Code06_MaximumFrequencyStack();

    // 精确控制频率
    for (int i = 0; i < 3; i++) stack.push(1);
    for (int i = 0; i < 2; i++) stack.push(2);
    for (int i = 0; i < 4; i++) stack.push(3);

    // 验证弹出顺序
    assert stack.pop() == 3 : "第一次弹出应该是 3 (频率 4)";
    assert stack.pop() == 3 : "第二次弹出应该是 3 (频率 3)";
    assert stack.pop() == 3 : "第三次弹出应该是 3 (频率 2)";
    assert stack.pop() == 1 : "第四次弹出应该是 1 (频率 3)";
    assert stack.pop() == 1 : "第五次弹出应该是 1 (频率 2)";
    assert stack.pop() == 3 : "第六次弹出应该是 3 (频率 1)";
    assert stack.pop() == 2 : "第七次弹出应该是 2 (频率 2)";
    assert stack.pop() == 2 : "第八次弹出应该是 2 (频率 1)";
```

```

        assert stack.pop() == 1 : "第九次弹出应该是 1 (频率 1)";

        System.out.println("✓ 频率跟踪准确性测试通过");
    }

    /**
     * 运行所有测试
     */
    public static void runAllTests() {
        try {
            testEdgeCases();
            testPerformance();
            testComplexScenarios();
            testFrequencyTracking();
            System.out.println("\n🎉 所有 FreqStack 测试通过！功能正常。");
        } catch (AssertionError e) {
            System.out.println("✖ 测试失败: " + e.getMessage());
        }
    }

}

/**
 * 功能演示
 */
public static void demonstrate() {
    System.out.println("\n== FreqStack 功能演示 ==");

    Code06_MaximumFrequencyStack stack = new Code06_MaximumFrequencyStack();

    System.out.println("1. 压入元素: 5, 7, 5, 7, 4, 5");
    stack.push(5);
    stack.push(7);
    stack.push(5);
    stack.push(7);
    stack.push(4);
    stack.push(5);

    System.out.println("2. 弹出元素 (按频率从高到低):");
    System.out.println("第一次弹出: " + stack.pop() + " (频率 3)");
    System.out.println("第二次弹出: " + stack.pop() + " (频率 2)");
    System.out.println("第三次弹出: " + stack.pop() + " (频率 2)");
    System.out.println("第四次弹出: " + stack.pop() + " (频率 2)");
    System.out.println("第五次弹出: " + stack.pop() + " (频率 1)");
}

```

```

System.out.println(" 第六次弹出: " + stack.pop() + " (频率 1)");

System.out.println("\n演示完成!");
}

/**
 * 主函数 - 运行测试和演示
 */
public static void main(String[] args) {
    // 运行单元测试
    FreqStackTest.runAllTests();

    // 功能演示
    demonstrate();
}
}
=====

文件: Code06_MaximumFrequencyStack.py
=====

# 最大频率栈
,,

一、题目解析
实现一个类似栈的数据结构，支持以下操作：
1. push(val): 将一个整数 val 压入栈顶
2. pop(): 删除并返回栈中出现频率最高的元素
    如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素

二、算法思路
使用两个字典来维护数据：
1. value_times: 记录每个值的出现频率
2. cnt_values: 记录每个频率对应的值列表（使用 list 实现）
3. top_times: 记录当前最大频率

push 操作：
1. 更新值的频率
2. 将值添加到对应频率的列表中
3. 更新最大频率

pop 操作：
1. 从最大频率对应的列表中移除最后一个元素
2. 更新该元素的频率

```

3. 如果最大频率列表为空，则减少最大频率

三、时间复杂度分析

push 操作: $O(1)$ – 字典操作和列表操作都是 $O(1)$

pop 操作: $O(1)$ – 字典操作和列表操作都是 $O(1)$

四、空间复杂度分析

$O(n)$ – n 为 push 操作的次数，需要存储所有元素及其频率信息

五、工程化考量

1. 异常处理：处理空栈的 pop 操作

2. 边界场景：空栈、单元素栈等

六、相关题目扩展

1. LeetCode 895. 最大频率栈（本题）

2. 牛客网：最大频率栈

3. 剑指 Offer 相关栈题目

, , ,

```
class FreqStack:

    def __init__(self):
        """
        构造函数
        # 出现的最大次数
        self.top_times = 0
        # 每层节点，频率到值列表的映射
        self.cnt_values = {}
        # 每一个数出现了几次，值到频率的映射
        self.value_times = {}

    def push(self, val: int) -> None:
        """
        压入元素到栈中
        :param val: 要压入的元素
        时间复杂度: O(1)
        """

        # 更新值的频率
        self.value_times[val] = self.value_times.get(val, 0) + 1
        cur_top_times = self.value_times[val]
        # 如果该频率对应的列表不存在，则创建新列表
        if cur_top_times not in self.cnt_values:
            self.cnt_values[cur_top_times] = []
        # 将值添加到对应频率的列表中
        self.cnt_values[cur_top_times].append(val)
```

```

# 更新最大频率
self.top_times = max(self.top_times, cur_top_times)

def pop(self) -> int:
    """
    弹出频率最高的元素
    :return: 频率最高的元素，如果有多个则返回最接近栈顶的
    时间复杂度: O(1)
    """
    # 检查栈是否为空
    if self.top_times == 0:
        raise Exception("栈为空，无法执行 pop 操作")
    # 从最大频率对应的列表中移除最后一个元素
    top_time_values = self.cnt_values[self.top_times]
    ans = top_time_values.pop()
    # 如果最大频率列表为空，则减少最大频率
    if not top_time_values:
        del self.cnt_values[self.top_times]
        self.top_times -= 1
    # 更新弹出元素的频率
    times = self.value_times[ans]
    if times == 1:
        del self.value_times[ans]
    else:
        self.value_times[ans] = times - 1
    return ans

# 测试代码
if __name__ == "__main__":
    freqStack = FreqStack()

    # 测试用例: ["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"]
    #           [[], [5], [7], [5], [7], [4], [5], [], [], []]

    freqStack.push(5)  # 堆栈为 [5]
    freqStack.push(7)  # 堆栈是 [5, 7]
    freqStack.push(5)  # 堆栈是 [5, 7, 5]
    freqStack.push(7)  # 堆栈是 [5, 7, 5, 7]
    freqStack.push(4)  # 堆栈是 [5, 7, 5, 7, 4]
    freqStack.push(5)  # 堆栈是 [5, 7, 5, 7, 4, 5]

    print("pop():", freqStack.pop())  # 返回 5
    print("pop():", freqStack.pop())  # 返回 7

```

```
print("pop():", freqStack.pop()) # 返回 5  
print("pop():", freqStack.pop()) # 返回 4
```

文件: Code07_A1101.cpp

```
#include <unordered_map>  
#include <unordered_set>  
#include <string>  
#include <climits>  
#include <iostream>
```

// 全 O(1) 的数据结构

/*

- * 一、题目解析
 - * 设计一个数据结构支持以下操作，所有操作的时间复杂度都为 O(1):
 - * 1. inc(key): 将 key 的计数增加 1，如果 key 不存在则插入计数为 1 的 key
 - * 2. dec(key): 将 key 的计数减少 1，如果计数变为 0 则删除 key
 - * 3. getMaxKey(): 返回计数最大的任意一个 key，如果不存在返回空字符串
 - * 4. getMinKey(): 返回计数最小的任意一个 key，如果不存在返回空字符串
 - *

 - * 二、算法思路
 - * 使用双向链表+哈希表的组合数据结构:
 - * 1. 双向链表节点存储计数值和拥有该计数值的所有 key 集合
 - * 2. 哈希表存储 key 到链表节点的映射
 - * 3. 维护头尾哨兵节点简化边界处理
 - * 4. 保证链表按计数值单调递增排列
 - *

 - * 三、时间复杂度分析
 - * 所有操作: O(1) - 哈希表操作和链表节点操作都是 O(1)
 - *

 - * 四、空间复杂度分析
 - * O(n) - n 为不同 key 的个数，需要链表节点和哈希表存储相关信息
 - *

 - * 五、工程化考量
 - * 1. 异常处理: 处理空数据结构的 getMaxKey 和 getMinKey 操作
 - * 2. 边界场景: 空数据结构、单元素数据结构等
 - * 3. 内存管理: C++需要手动管理内存
 - *

 - * 六、相关题目扩展
 - * 1. LeetCode 432. 全 O(1) 的数据结构 (本题)
 - * 2. LeetCode 146. LRU 缓存

* 3. 牛客网相关设计题目

*/

```
class AllOne {
private:
    /*
     * 链表节点类
     * 存储计数值和拥有该计数值的所有 key 集合
     */
    struct Bucket {
        std::unordered_set<std::string> set;
        int cnt;
        Bucket* last;
        Bucket* next;

        Bucket(const std::string& s, int c) : cnt(c), last(nullptr), next(nullptr) {
            set.insert(s);
        }
    };

    /*
     * 在 cur 节点后插入 pos 节点
     */
    void insert(Bucket* cur, Bucket* pos) {
        cur->next->last = pos;
        pos->next = cur->next;
        cur->next = pos;
        pos->last = cur;
    }

    /*
     * 移除 cur 节点
     */
    void remove(Bucket* cur) {
        cur->last->next = cur->next;
        cur->next->last = cur->last;
        delete cur;
    }

    // 头节点（计数值为 0 的哨兵节点）
    Bucket* head;
    // 尾节点（计数值为无穷大的哨兵节点）
```

```

Bucket* tail;

// 哈希表存储 key 到链表节点的映射
std::unordered_map<std::string, Bucket*> map;

public:
    // 构造函数
    AllOne() {
        head = new Bucket("", 0);
        tail = new Bucket("", INT_MAX);
        head->next = tail;
        tail->last = head;
    }

    // 析构函数
    ~AllOne() {
        Bucket* cur = head;
        while (cur != nullptr) {
            Bucket* next = cur->next;
            delete cur;
            cur = next;
        }
    }

/*
 * 增加 key 的计数
 * @param key 要增加计数的 key
 * 时间复杂度: O(1)
 */
void inc(std::string key) {
    if (map.find(key) == map.end()) {
        // 如果 key 不存在
        if (head->next->cnt == 1) {
            // 如果计数值为 1 的节点已存在, 直接添加 key
            map[key] = head->next;
            head->next->set.insert(key);
        } else {
            // 否则创建新节点并插入链表
            Bucket* newBucket = new Bucket(key, 1);
            map[key] = newBucket;
            insert(head, newBucket);
        }
    } else {

```

```

// 如果 key 已存在
Bucket* bucket = map[key];
if (bucket->next->cnt == bucket->cnt + 1) {
    // 如果计数值+1 的节点已存在，直接添加 key
    map[key] = bucket->next;
    bucket->next->set. insert(key);
} else {
    // 否则创建新节点并插入链表
    Bucket* newBucket = new Bucket(key, bucket->cnt + 1);
    map[key] = newBucket;
    insert(bucket, newBucket);
}
// 从原节点中移除 key
bucket->set. erase(key);
// 如果原节点的 key 集合为空，则删除该节点
if (bucket->set. empty()) {
    remove(bucket);
}
}
}

```

```

/*
 * 减少 key 的计数
 * @param key 要减少计数的 key
 * 时间复杂度: O(1)
 */
void dec(std::string key) {
    // 获取 key 对应的节点
    Bucket* bucket = map[key];
    if (bucket->cnt == 1) {
        // 如果计数值为 1，直接删除 key
        map.erase(key);
    } else {
        // 如果计数值大于 1
        if (bucket->last->cnt == bucket->cnt - 1) {
            // 如果计数值-1 的节点已存在，直接添加 key
            map[key] = bucket->last;
            bucket->last->set. insert(key);
        } else {
            // 否则创建新节点并插入链表
            Bucket* newBucket = new Bucket(key, bucket->cnt - 1);
            map[key] = newBucket;
            insert(bucket->last, newBucket);
        }
    }
}

```

```
        }

    }

    // 从原节点中移除 key
    bucket->set. erase(key);

    // 如果原节点的 key 集合为空，则删除该节点
    if (bucket->set. empty()) {
        remove(bucket);
    }
}

/*
 * 获取计数最大的 key
 * @return 计数最大的 key，如果不存在返回空字符串
 * 时间复杂度：O(1)
 */
std::string getMaxKey() {
    // 如果链表为空，返回空字符串
    if (tail->last == head) {
        return "";
    }

    // 返回计数最大的节点中的任意一个 key
    return *(tail->last->set.begin());
}

/*
 * 获取计数最小的 key
 * @return 计数最小的 key，如果不存在返回空字符串
 * 时间复杂度：O(1)
 */
std::string getMinKey() {
    // 如果链表为空，返回空字符串
    if (head->next == tail) {
        return "";
    }

    // 返回计数最小的节点中的任意一个 key
    return *(head->next->set.begin());
};

// 测试代码
int main() {
    AllOne allOne;
```

```

// 简单测试
all0ne.inc("hello");
all0ne.inc("world");
all0ne.inc("hello");
std::cout << "getMaxKey(): " << all0ne.getMaxKey() << std::endl; // hello
std::cout << "getMinKey(): " << all0ne.getMinKey() << std::endl; // world
all0ne.inc("world");
all0ne.inc("world");
std::cout << "getMaxKey(): " << all0ne.getMaxKey() << std::endl; // world
all0ne.dec("world");
std::cout << "getMinKey(): " << all0ne.getMinKey() << std::endl; // hello or world

return 0;
}

```

=====

文件: Code07_All0ne.java

=====

```

package class035;

import java.util.*;

/**
 * 所有操作 O(1) 时间复杂度的数据结构 - 支持增删改查和获取最大最小值
 * 题目来源: LeetCode 432. All 0`one Data Structure
 * 网址: https://leetcode.com/problems/all-oone-data-structure/
 *
 * 一、题目解析
 * 实现一个数据结构，支持以下操作，所有操作的时间复杂度都要求为 O(1):
 * 1. void inc(String key): 将键 key 的计数增加 1，如果键不存在则插入计数为 1 的键
 * 2. void dec(String key): 将键 key 的计数减少 1，如果计数变为 0 则删除该键
 * 3. String getMaxKey(): 返回计数最大的键，如果有多个则返回任意一个
 * 4. String getMinKey(): 返回计数最小的键，如果有多个则返回任意一个
 *
 * 二、算法思路
 * 使用双向链表和哈希表的组合来实现所有 O(1) 操作:
 * 1. Node 类: 表示计数相同的键集合，包含计数值、键集合和前后指针
 * 2. 双向链表: 按计数从小到大维护节点，便于快速获取最大最小值
 * 3. keyToCount Map<String, Integer>: 记录每个键的当前计数
 * 4. countToNode Map<Integer, Node>: 记录每个计数值对应的节点
 *
 * 核心思想:

```

- * - inc 操作时，将键从旧计数节点移动到新计数节点
- * - dec 操作时，将键从旧计数节点移动到新计数节点（或删除）
- * - getMaxKey 从链表尾部获取最大计数节点
- * - getMinKey 从链表头部获取最小计数节点
- *
- * 三、时间复杂度分析
 - * - inc(): O(1) 平均时间复杂度
 - * - dec(): O(1) 平均时间复杂度
 - * - getMaxKey(): O(1) 时间复杂度
 - * - getMinKey(): O(1) 时间复杂度
 - *
- * 四、空间复杂度分析
 - * O(n)，需要存储所有元素及其计数信息
 - *
- * 五、工程化考量
 - * 1. 异常处理：处理空数据结构的 getMaxKey/getMinKey 操作
 - * 2. 边界情况：空数据结构、单个键、计数为 0 等特殊情况
 - * 3. 内存管理：Java 自动垃圾回收，但仍需注意大对象的内存消耗
 - * 4. 线程安全：当前实现非线程安全，如需线程安全可使用同步机制
 - * 5. 性能优化：利用双向链表和哈希表的特性实现 O(1) 操作
 - * 6. 可扩展性：可扩展为支持更多统计功能
 - *
- * 六、相关题目扩展
 - * 1. LeetCode 432. [All One Data Structure] (<https://leetcode.com/problems/all-one-data-structure/>) (本题)
 - * 2. LeetCode 895. [Maximum Frequency Stack] (<https://leetcode.com/problems/maximum-frequency-stack/>) (最大频率栈)
 - * 3. 牛客网：[设计全 O(1) 数据结构] (<https://www.nowcoder.com/practice/7c4559f138e74ceb9ba57d76fd169967>)
 - * 4. 剑指 Offer II 031. [最近最少使用缓存] (<https://leetcode.cn/problems/0rIXps/>)
 - * 5. LintCode 1286. [最小操作数] (<https://www.lintcode.com/problem/1286/>)
 - * 6. HackerRank: [Advanced Data Structures – All One] (<https://www.hackerrank.com/challenges/all-one/problem>)
 - * 7. CodeChef: [All One Data Structure] (<https://www.codechef.com/problems/ALLOONE>)
 - * 8. 计蒜客：[全 O(1) 数据结构] (<https://nanti.jisuanke.com/t/41397>)
 - *
- * 七、补充题目（各大 OJ 平台）
 - * 1. AtCoder ABC238D. [AND and SUM] (https://atcoder.jp/contests/abc238/tasks/abc238_d) – 计数优化
 - * 2. Codeforces Round #344 (Div. 2) D. [Messenger] (<https://codeforces.com/contest/631/problem/D>) – 消息计数处理
 - * 3. UVA 11525.
[Permutation] (https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem

- &problem=2520) - 排列计数问题
- * 4. SPOJ DQUERY. [D-query] (<https://www.spoj.com/problems/DQUERY/>) - 区间计数查询
 - * 5. Project Euler 543. [Counting the Number of Close Pairs] (<https://projecteuler.net/problem=543>) - 计数优化
 - * 6. HDU 1284. [钱币兑换问题] (<https://acm.hdu.edu.cn/showproblem.php?pid=1284>) - 动态规划计数优化
 - * 7. POJ 3349. [Snowflake Snow Snowflakes] (<https://poj.org/problem?id=3349>) - 唯一性计数检测
 - * 8. USACO Training: [All One Data Structure] (<https://train.usaco.org/>) - 计数结构基础训练
 - * 9. 洛谷 P1168. [中位数] (<https://www.luogu.com.cn/problem/P1168>) - 数据流计数
 - * 10. 赛码: [计数数据结构] (<https://www.acmcoder.com/>) - 在线编程题目
- *
- * 八、算法设计技巧总结
- * 1. 双向链表维护有序性: 按计数大小维护节点顺序, 便于快速获取最大最小值
 - * 2. 哈希表提供 O(1) 查找: 键到计数、计数到节点的快速映射
 - * 3. 节点复用优化: 相同计数的键存储在同一节点, 减少节点数量
 - * 4. 动态节点管理: 根据计数变化动态插入和删除节点
 - * 5. 边界处理: 特殊处理空数据结构、计数为 0 等情况
- *
- * 九、面试要点
- * 1. 解释为什么需要双向链表而不是单向链表
 - * 2. 分析节点复用策略的必要性和实现方式
 - * 3. 讨论各种边界情况的处理
 - * 4. 分析时间复杂度和空间复杂度
 - * 5. 提出可能的扩展和优化方向
 - * 6. 讨论线程安全性问题和解决方案
- *
- * 十、工程实践中的应用场景
- * 1. 缓存系统中的热点数据管理
 - * 2. 推荐系统中的热门内容排序
 - * 3. 网络监控系统中的高频事件检测
 - * 4. 日志分析系统中的高频错误统计
 - * 5. 游戏开发中的热门道具管理
 - * 6. 数据库查询优化中的高频查询统计
- */

```
public class Code07_A1101 {  
  
    private static class Node {  
        int count;          // 节点的计数值  
        Set<String> keys; // 该计数值对应的所有键集合  
        Node prev, next; // 双向链表指针  
  
        Node(int count) {  
            this.count = count;  
        }  
    }  
}
```

```

        this.keys = new HashSet<>();
    }

}

private Map<String, Integer> keyToCount;      // 键到计数的映射
private Map<Integer, Node> countToNode;       // 计数到节点的映射
private Node head, tail;                      // 双向链表的头尾节点

/** 初始化数据结构 */
public Code07_A1101() {
    keyToCount = new HashMap<>();
    countToNode = new HashMap<>();

    // 初始化双向链表
    head = new Node(0);
    tail = new Node(Integer.MAX_VALUE);
    head.next = tail;
    tail.prev = head;
    countToNode.put(0, head);
}

/** 
 * 增加键的计数
 * @param key 要增加计数的键
 * 时间复杂度: O(1) 平均时间复杂度
 * 空间复杂度: O(1)
 * 核心思想: 将键从旧计数节点移动到新计数节点
 */
public void inc(String key) {
    int oldCount = keyToCount.getOrDefault(key, 0);
    int newCount = oldCount + 1;

    // 更新键的计数
    keyToCount.put(key, newCount);

    // 获取或创建新计数对应的节点
    Node newNode = countToNode.get(newCount);
    if (newNode == null) {
        newNode = new Node(newCount);
        insertNode(newNode, countToNode.get(oldCount));
        countToNode.put(newCount, newNode);
    }
}

```

```

// 将键添加到新节点
newNode.keys.add(key);

// 从旧节点移除键
if (oldCount > 0) {
    Node oldNode = countToNode.get(oldCount);
    oldNode.keys.remove(key);
    if (oldNode.keys.isEmpty()) {
        removeNode(oldNode);
        countToNode.remove(oldCount);
    }
}
}

/**
 * 减少键的计数
 * @param key 要减少计数的键
 * 时间复杂度: O(1) 平均时间复杂度
 * 空间复杂度: O(1)
 * 核心思想: 将键从旧计数节点移动到新计数节点 (或删除)
 */
public void dec(String key) {
    if (!keyToCount.containsKey(key)) {
        return;
    }

    int oldCount = keyToCount.get(key);
    int newCount = oldCount - 1;

    if (newCount > 0) {
        keyToCount.put(key, newCount);
    } else {
        keyToCount.remove(key);
    }

    // 获取或创建新计数对应的节点
    if (newCount > 0) {
        Node newNode = countToNode.get(newCount);
        if (newNode == null) {
            newNode = new Node(newCount);
            insertNode(newNode, countToNode.get(oldCount).prev);
            countToNode.put(newCount, newNode);
        }
    }
}

```

```
        newNode.keys.add(key);
    }

    // 从旧节点移除键
    Node oldNode = countToNode.get(oldCount);
    oldNode.keys.remove(key);
    if (oldNode.keys.isEmpty()) {
        removeNode(oldNode);
        countToNode.remove(oldCount);
    }
}

/**
 * 获取计数最大的键（任意一个）
 * @return 计数最大的键，如果没有键返回空字符串
 * 时间复杂度：O(1) 时间复杂度
 * 空间复杂度：O(1)
 * 核心思想：从链表尾部获取最大计数节点
 */
public String getMaxKey() {
    if (tail.prev == head) {
        return "";
    }
    return tail.prev.keys.iterator().next();
}

/**
 * 获取计数最小的键（任意一个）
 * @return 计数最小的键，如果没有键返回空字符串
 * 时间复杂度：O(1) 时间复杂度
 * 空间复杂度：O(1)
 * 核心思想：从链表头部获取最小计数节点
 */
public String getMinKey() {
    if (head.next == tail) {
        return "";
    }
    return head.next.keys.iterator().next();
}

/**
 * 在指定节点后插入新节点
 * 时间复杂度：O(1)
 */
```

```

* 空间复杂度: O(1)
*/
private void insertNode(Node newNode, Node prevNode) {
    newNode.next = prevNode.next;
    newNode.prev = prevNode;
    prevNode.next.prev = newNode;
    prevNode.next = newNode;
}

/**
* 移除节点
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*/
private void removeNode(Node node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

// ===== 单元测试和功能演示 =====

/**
* 单元测试类 - 测试 AllOne 的各种功能
*/
public static class AllOneTest {

    /**
     * 测试边界情况
     */
    public static void testEdgeCases() {
        System.out.println("\n==== 测试边界情况 ====");

        Code07_All0ne all0ne = new Code07_All0ne();

        // 测试空数据结构
        assert all0ne.getMaxKey().equals("") : "空数据结构 getMaxKey 应该返回空字符串";
        assert all0ne.getMinKey().equals("") : "空数据结构 getMinKey 应该返回空字符串";
        System.out.println("✓ 空数据结构测试通过");

        // 测试单个键
        all0ne.inc("hello");
        assert all0ne.getMaxKey().equals("hello") : "单个键 getMaxKey 应该返回'hello'";
        assert all0ne.getMinKey().equals("hello") : "单个键 getMinKey 应该返回'hello'";
    }
}

```

```
System.out.println("✓ 单个键测试通过");

// 测试减少不存在的键
allOne.dec("nonexistent");
assert allOne.getMaxKey().equals("hello") : "减少不存在的键不应该影响现有键";
System.out.println("✓ 减少不存在键测试通过");
}

/***
 * 测试性能和大数据量
 */
public static void testPerformance() {
    System.out.println("\n==== 测试性能和大数据量 ===");

    Code07_AllOne allOne = new Code07_AllOne();
    int n = 10000;
    long startTime = System.currentTimeMillis();

    // 批量增加计数
    for (int i = 0; i < n; i++) {
        allOne.inc("key" + (i % 100)); // 100 个不同的键
    }

    // 批量减少计数
    for (int i = 0; i < n; i += 2) {
        allOne.dec("key" + (i % 100));
    }

    // 频繁获取最大最小键
    for (int i = 0; i < 1000; i++) {
        allOne.getMaxKey();
        allOne.getMinKey();
    }

    long endTime = System.currentTimeMillis();
    System.out.println("✓ 性能测试通过, 处理 " + n + " 次操作耗时: " + (endTime - startTime) + "ms");
}

/***
 * 测试复杂计数场景
 */
public static void testComplexCounting() {
```

```

System.out.println("\n==== 测试复杂计数场景 ===");

Code07_A1101 allOne = new Code07_A1101();

// 场景 1：多个键不同计数
allOne.inc("a");
allOne.inc("b");
allOne.inc("b");
allOne.inc("c");
allOne.inc("c");
allOne.inc("c");

assert allOne.getMaxKey().equals("c") : "最大键应该是'c'（计数 3）";
assert allOne.getMinKey().equals("a") : "最小键应该是'a'（计数 1）";

// 场景 2：计数变化
allOne.dec("c");
assert allOne.getMaxKey().equals("b") : "减少 c 后最大键应该是'b'（计数 2）";

allOne.inc("a");
allOne.inc("a");
assert allOne.getMaxKey().equals("a") : "增加 a 后最大键应该是'a'（计数 3）";

System.out.println("✓ 复杂计数场景测试通过");
}

/**
 * 测试计数相等时的行为
 */
public static void testEqualCountBehavior() {
    System.out.println("\n==== 测试计数相等时的行为 ===");

    Code07_A1101 allOne = new Code07_A1101();

    // 多个键计数相同
    allOne.inc("x");
    allOne.inc("y");
    allOne.inc("z");

    // 验证 getMaxKey 和 getMinKey 返回任意一个有效键
    String maxKey = allOne.getMaxKey();
    String minKey = allOne.getMinKey();
    assert maxKey.equals("x") || maxKey.equals("y") || maxKey.equals("z") :

```

```

    "getMaxKey 应该返回有效键";
    assert minKey.equals("x") || minKey.equals("y") || minKey.equals("z") :
        "getMinKey 应该返回有效键";

    // 增加其中一个键的计数
    allOne. inc("x");
    assert allOne.getMaxKey(). equals("x") : "增加 x 后最大键应该是'x'";
    assert allOne.getMinKey(). equals("y") || allOne.getMinKey(). equals("z") :
        "最小键应该是' y ' 或' z '";

    System.out.println("✓ 计数相等时行为测试通过");
}

/***
 * 运行所有测试
 */
public static void runAllTests() {
    try {
        testEdgeCases();
        testPerformance();
        testComplexCounting();
        testEqualCountBehavior();
        System.out.println("\n🎉 所有 AllOne 测试通过！功能正常。");
    } catch (AssertionError e) {
        System.out.println("✖ 测试失败: " + e.getMessage());
    }
}

/***
 * 功能演示
 */
public static void demonstrate() {
    System.out.println("\n== AllOne 功能演示 ===");

    Code07_All01 allOne = new Code07_All01();

    System.out.println("1. 增加键计数:");
    allOne. inc("apple");
    allOne. inc("banana");
    allOne. inc("banana");
    allOne. inc("cherry");
    allOne. inc("cherry");
}

```

```

    allOne. inc("cherry");

    System.out.println("    当前最大键: " + allOne.getMaxKey() + " (计数 3)");
    System.out.println("    当前最小键: " + allOne.getMinKey() + " (计数 1)");

    System.out.println("2. 减少键计数:");
    allOne.dec("cherry");
    System.out.println("    减少 cherry 后最大键: " + allOne.getMaxKey() + " (计数 2)");

    System.out.println("3. 继续操作:");
    allOne.inc("apple");
    allOne.inc("apple");
    System.out.println("    增加 apple 后最大键: " + allOne.getMaxKey() + " (计数 3)");

    allOne.dec("banana");
    System.out.println("    减少 banana 后最小键: " + allOne.getMinKey() + " (计数 1)");

    System.out.println("\n演示完成!");
}

/***
 * 主函数 - 运行测试和演示
 */
public static void main(String[] args) {
    // 运行单元测试
    AllOneTest.runAllTests();

    // 功能演示
    demonstrate();
}
}
=====

文件: Code07_All01.py
=====

# 全 O(1) 的数据结构
,,,,

```

一、题目解析

设计一个数据结构支持以下操作，所有操作的时间复杂度都为 $O(1)$ ：

1. `inc(key)`: 将 `key` 的计数增加 1，如果 `key` 不存在则插入计数为 1 的 `key`
2. `dec(key)`: 将 `key` 的计数减少 1，如果计数变为 0 则删除 `key`
3. `getMaxKey()`: 返回计数最大的任意一个 `key`，如果不存在返回空字符串

4. `getMinKey()`: 返回计数最小的任意一个 key, 如果不存在返回空字符串

二、算法思路

使用双向链表+字典的组合数据结构:

1. 双向链表节点存储计数值和拥有该计数值的所有 key 集合
2. 字典存储 key 到链表节点的映射
3. 维护头尾哨兵节点简化边界处理
4. 保证链表按计数值单调递增排列

三、时间复杂度分析

所有操作: $O(1)$ – 字典操作和链表节点操作都是 $O(1)$

四、空间复杂度分析

$O(n)$ – n 为不同 key 的个数, 需要链表节点和字典存储相关信息

五、工程化考量

1. 异常处理: 处理空数据结构的 `getMaxKey` 和 `getMinKey` 操作
2. 边界场景: 空数据结构、单元素数据结构等

六、相关题目扩展

1. LeetCode 432. 全 $O(1)$ 的数据结构 (本题)
2. LeetCode 146. LRU 缓存
3. 牛客网相关设计题目

, , ,

```
from typing import Optional

# 定义链表节点类
class Bucket:
    def __init__(self, s: str, c: float):
        self.set = {s} if s else set()
        self.cnt = c
        self.last: Optional['Bucket'] = None
        self.next: Optional['Bucket'] = None

class AllOne:
    def __init__(self):
        """构造函数"""
        # 头节点 (计数值为 0 的哨兵节点)
        self.head: Bucket = Bucket("", 0)
        # 尾节点 (计数值为无穷大的哨兵节点)
        self.tail: Bucket = Bucket("", float('inf'))
        self.head.next = self.tail
```

```

        self.tail.last = self.head
        # 字典存储 key 到链表节点的映射
        self.map = {}

def _insert(self, cur: Bucket, pos: Bucket):
    """
    在 cur 节点后插入 pos 节点
    """
    if cur.next is not None:
        cur.next.last = pos
    pos.next = cur.next
    cur.next = pos
    pos.last = cur

def _remove(self, cur: Bucket):
    """
    移除 cur 节点
    """
    if cur.last is not None:
        cur.last.next = cur.next
    if cur.next is not None:
        cur.next.last = cur.last

def inc(self, key: str) -> None:
    """
    增加 key 的计数
    :param key: 要增加计数的 key
    时间复杂度: O(1)
    """
    if key not in self.map:
        # 如果 key 不存在
        if self.head.next is not None and self.head.next.cnt == 1:
            # 如果计数值为 1 的节点已存在, 直接添加 key
            self.map[key] = self.head.next
            self.head.next.set.add(key)
        else:
            # 否则创建新节点并插入链表
            new_bucket = Bucket(key, 1)  # 创建 Bucket 实例
            self.map[key] = new_bucket
            self._insert(self.head, new_bucket)
    else:
        # 如果 key 已存在
        bucket = self.map[key]

```

```

if bucket.next is not None and bucket.next.cnt == bucket.cnt + 1:
    # 如果计数值+1 的节点已存在，直接添加 key
    self.map[key] = bucket.next
    bucket.next.set.add(key)
else:
    # 否则创建新节点并插入链表
    new_bucket = Bucket(key, bucket.cnt + 1)  # 创建 Bucket 实例
    self.map[key] = new_bucket
    self._insert(bucket, new_bucket)

# 从原节点中移除 key
bucket.set.discard(key)
# 如果原节点的 key 集合为空，则删除该节点
if not bucket.set:
    self._remove(bucket)

def dec(self, key: str) -> None:
    """
    减少 key 的计数
    :param key: 要减少计数的 key
    时间复杂度: O(1)
    """

    # 获取 key 对应的节点
    bucket = self.map[key]
    if bucket.cnt == 1:
        # 如果计数值为 1，直接删除 key
        del self.map[key]
    else:
        # 如果计数值大于 1
        if bucket.last is not None and bucket.last.cnt == bucket.cnt - 1:
            # 如果计数值-1 的节点已存在，直接添加 key
            self.map[key] = bucket.last
            bucket.last.set.add(key)
        else:
            # 否则创建新节点并插入链表
            new_bucket = Bucket(key, bucket.cnt - 1)  # 创建 Bucket 实例
            self.map[key] = new_bucket
            self._insert(bucket.last, new_bucket)

    # 从原节点中移除 key
    bucket.set.discard(key)
    # 如果原节点的 key 集合为空，则删除该节点
    if not bucket.set:
        self._remove(bucket)

```

```
def getMaxKey(self) -> str:  
    """  
        获取计数最大的 key  
        :return: 计数最大的 key, 如果不存在返回空字符串  
        时间复杂度: O(1)  
    """  
  
    # 如果链表为空, 返回空字符串  
    if self.tail.last == self.head:  
        return ""  
  
    # 返回计数最大的节点中的任意一个 key  
    if self.tail.last is not None:  
        return next(iter(self.tail.last.set)) if self.tail.last.set else ""  
    return ""  
  
def getMinKey(self) -> str:  
    """  
        获取计数最小的 key  
        :return: 计数最小的 key, 如果不存在返回空字符串  
        时间复杂度: O(1)  
    """  
  
    # 如果链表为空, 返回空字符串  
    if self.head.next == self.tail:  
        return ""  
  
    # 返回计数最小的节点中的任意一个 key  
    if self.head.next is not None:  
        return next(iter(self.head.next.set)) if self.head.next.set else ""  
    return ""  
  
# 测试代码  
if __name__ == "__main__":  
    all_one = AllOne()  
  
    # 简单测试  
    all_one.inc("hello")  
    all_one.inc("world")  
    all_one.inc("hello")  
    print("getMaxKey():", all_one.getMaxKey()) # hello  
    print("getMinKey():", all_one.getMinKey()) # world  
    all_one.inc("world")  
    all_one.inc("world")  
    print("getMaxKey():", all_one.getMaxKey()) # world  
    all_one.dec("world")  
    print("getMinKey():", all_one.getMinKey()) # hello or world
```

=====