LinkedHashMap实现及在LRU算法中的应用

蒋捷

2024 年 2 月 21 日

摘要

作业内容：提供LinkedHashMap的接口和LRU(Least Recently Used)的函数接口，要求实现LinkedHashMap的功能并以此为基础实现LRU算法。所有实现的功能都要求期望复杂度是 O(1)

作业限制：可以使用的头文件包含

截止时间：

提交方式：

评分标准：

目录

1 概念介绍

1.1 HashMap

LinkedHashMap在本质上是一个实现了按照插入顺序访问元素的HashMap(哈希表)，所以在介绍 LinkedHashMap之前，我们需要先了解HashMap。

1.1.1 哈希表相关概念

哈希值：就是把任意长度的输入，通过某种哈希算法，变换某种与之对应的输出 (通常是整数)。

哈希碰撞：不同的输入可能会散列成相同的输出，从而不可能从散列值来唯一的确定输入值。

(在本次作业中，推荐使用 std::hash 来实现哈希函数)

HashMap 支持关键词对应元素的插入、查询和删除，并且这些操作的平均时间复杂度都为$O(1)$，只会在最差情况下退化为$O(n)$。

1.1.2 固定大小哈希表

有了哈希函数，我们最自然的想法就是构建一个长度为l 的数组，数组的下标对应着哈希值。每当我们插入一对键值对(K, V) 的时候，我们先通过哈希函数对K 进行处理得到哈希值h，然后将v储存到数组的第$h\%l$ 位，但是我们会发现一个问题，当遇到之前所说的哈希碰撞问题的时候，我们没有办法在数组的同一个下标处储存两个元素，那么比较自然的想法对于每一个下标我们都开一个链表，如果遇到冲突问题，我们就在链表的末尾添加这个元素。我们查询的时候对于一个给定的K 首先先进行哈希得到哈希值h，再在第$h\%l$ 个链表中依次寻找K 所对应的元素，这就实现了一个固定大小的哈希表。

1.1.3 动态大小哈希表

我们需要实现的是一个封装好的数据结构供其他人使用，我们在开发的时候实际上并不知道数据规模的大小，所以对于哈希表大小的选择是一个非常关键的问题，当我们选择的哈希表大小比较小的时候，链表长度可能会比较长，查询复杂度会退化，当哈希表大小比较大的时候，会占用很多无用空间。这也启发我们可以动态的改变哈希表的大小。

首先引入两个参数C, f，分别是容量(Capacity)和负载因子(LoadFactor)，代表着哈希表的大小，和对于某个特定的容量，我们所能接受的最多元素个数占容量的比例。我们一开始可以选择一个比较小的容量，当元素个数大于$C * f$ 时，我们再增大C，使得我们的数据结构

能够保持良好效率的同时，不占用过多空间，具体参数大家可以根据自己实现的数据结构去进行调整。

1.2 LinkedHashMap

LinkedHashMap需要在实现HashMap功能的基础上，再进行插入顺序的维护，使得我们可以按照插入顺序来访问元素。这项功能的实现比较简单，只需要维护一个双向链表，每次添加的元素除了插入 HashMap外，也需要插入到链表的末尾，这样我们按照链表顺序访问元素就可以实现这项功能。

1.3 LRU

1.3.1 LRU相关概念

LRU(Least Recently Used)即最近最少使用算法，是一种内存数据淘汰策略，使用常见场景是当内存不足时，需要淘汰最近最少使用(被插入或者被查询)的数据。（注：内存可以理解为一个有限长度的数组，正如数组由许多元素组成，内存也包含很多内存空间。）

因此，该算法有一个参数n 表示预设的内存的大小（允许存储的键值对的个数）。

1.3.2 LRU执行的操作

- **插入(save)**

  可以理解为把一个键值对(K, V) 放入内存。首先查找有无内存空间的键为K

  - 有

    更新节点的值

    选取某空间标记为被该K' 使用并把存的值更新为V

  - 无

    检查有无未被使用的内存空间

    * 有

      将该内存空间标记为被该K 使用并存入V

    * 无

      查找最早**被插入或者被查询**的键值对(K', V')，将该键值对替换为(K, V) (原键、值都不再存在)

- **查询(get)**

  利用Hash查找内存中有无某个内存空间该键K

  - 没找到，返回空。
  - 找到，返回缓存的值V

实现关键在于利用LinkedHashMap维护最早**被插入或者被查询**的键值对，即LRU。

2 项目文件