《数据结构》实验报告

班　级: 0601200　　姓　　名: 薛茵午　　学　号: 2020303828
E-mail: 814680344@qq.com　　日　　期: 2023/4/16

◎实验题目:　自适应缓存替换算法的双链表实现

◎实验内容:

使用循环双链表实现自适应缓存替换(ARC)算法并实现单词缓存功能

一、需求分析

本程序的灵感来源于力扣题目 460(LFU 缓存)， 对于 LFU 算法而言，其核心思想是在缓存中维护每一个词汇的使用频率，并且在缓存满时，移除最不经常使用的元素。如果有两个相同使用频率的项，则移除最晚使用过的元素。

在中文输入法和各种 IDE 的代码补全功能中，为了保证代码补全功能实现，每一次需要将最近使用频繁的元素存放在缓存中，然后在用户输入时，从缓存中搜索最近使用过的单词然后提供给用户进行补全，而中文输入法也是往往会将使用频率高的词进行前置供用户输入。

对于不同种类的缓存算法而言，为了保证查找和插入删除的快速性，往往使用双链表和哈希表作为存储结构，并且结合哈希表的快速查找特性和双链表的容易插入和删除的特性，实现数据的高速缓存。

对于不同的缓存算法，其各有优势和缺陷，目前常用的算法中，LFU(最近最不常用算法)可以按照使用频率对数据进行淘汰，而 LRU(最近最少使用算法)是按照数据的历史访问时间来淘汰数据的。由于数据存储特点不同，两种算法都会出现缓存污染现象。其中 LFU 对于突然某一数据短时间内大量使用而之后长时间内不用，往往会导致缓存占用，而 LRU 对于突然多种不同的数据使用，会覆盖原始的常用数据。

ARC 算法(Adaptive Replacement Cache,自适应缓存替换算法)是美国 IBM Almaden 研究中心开发的一种缓存算法,这种算法结合了 LRU 和 LFU 的各自的优点,可实现高效的缓存。

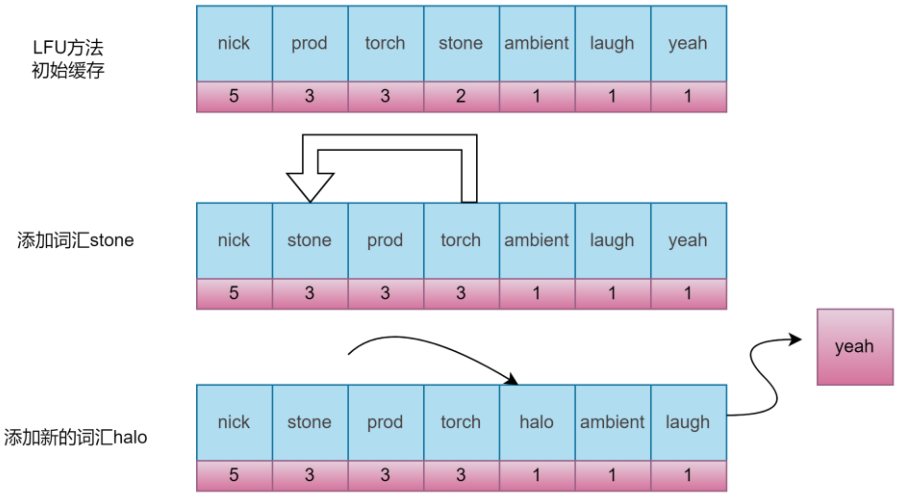
本程序使用 ARC 算法实现代码自动补全类型的缓存结构进行缓存,实现类似于输入法和 IDE 代码补全中的缓存功能。

二、实现流程

ARC 算法是将 LRU 缓存算法和 LFU 缓存算法结合起来的一种算法

我们以词汇的缓存为例分别说明 LRU, LFU 和 ARC 算法。首先，LRU 算法是一种仅存储最近使用过词语的一种存储结构，其原理是利用链表的有序性，在使用词汇时，如果使用的词汇在缓存中，则对最近使用的词汇进行前置，而每一次缓存区满时，我们会删除最长时间没有使用过的节点(如下图中，python 被删除)，并将新词语插入到链表的头部。

而对于 LFU 算法，则是在节点中使用一个关键词 time，存储对应的词汇被调用的次数。访问次数越大的，则越靠前，每一次加入新词汇时，删除其中访问次数最少的元素，如果其中有多个访问次数最少的元素，则删除其中距离现在调用时间最长的元素

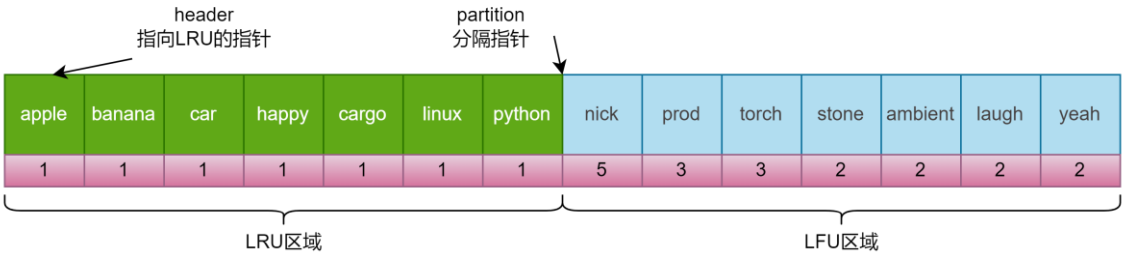


LRU 缓存淘汰算法和 LFU 缓存淘汰算法都是有其缺陷的，对于 LRU 算法，由于仅记录访问元素的顺序，对于我们常用的高频词组难以保存。尤其是当在短时间内大量访问新数据的情况，往往会将我们常用的高频词语驱逐出缓存，而导致补全效果下降。而对于 LFU 算法，如果某个生僻的词语在短时间内受到大量使用，则之后很难将其逐出缓存。上述两种缓存淘汰算法均会产生缓存利用不充分的现象，即缓存污染现象。

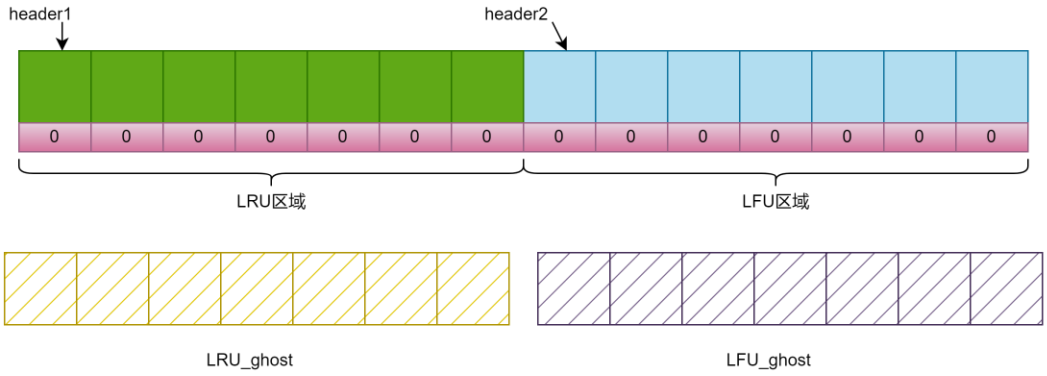
分别分析 LRU 和 LFU 的优缺点，我们为了能够充分利用缓存空间，我们可以将一个固定大小的缓存区分为 LRU 缓存区和 LFU 缓存区两个部分，如下图所示。这也是 ARC 算法的基本思想。

其动态分配内存的思想是，在输入重复数据较多时，LFU 区域长度增加，存储更多的高频内容；而在多次输入不同的数据时，LRU 内存区域增加，这样就可以实现内存的动态分配了。

在下图中，Partition 为分隔指针，为了能够实现内存的充分使用，Partition 指针的位置可以进行动态调整，也就是说，LRU 和 LFU 的区域大小可以进行动态改变，实现动态分配内存，分别存储最近使用和最频繁使用的数据。而在程序中，我没有使用 partition,而是使用两个长度之和为定值的双链表进行。

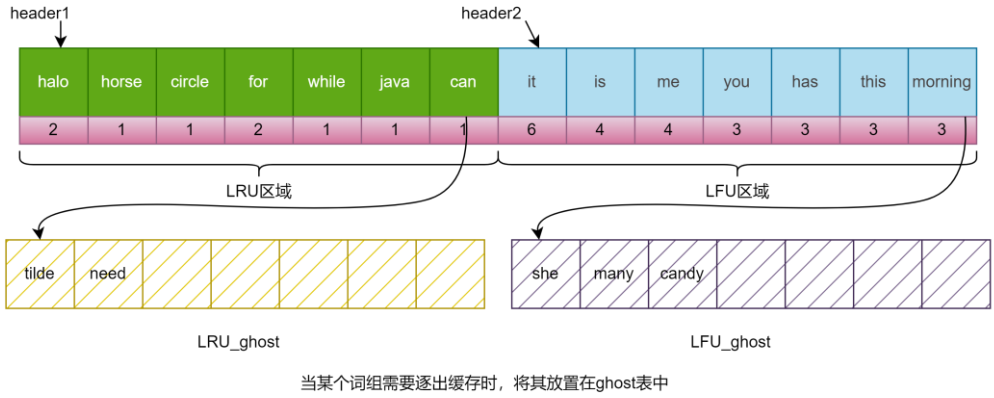


其中，较为重要的是如何实现上述两个表的大小动态调整。我们可以对 LRU 区域和 LFU 区域分别建立一个淘汰链表(或者称为虚表)，分别存储 LRU 和 LFU 中被淘汰的元素。这两个链表分别称为 LRU_ghost 和 LFU_ghost,在实际使用中为了节省内存，ghost 表部分只存储键值，而不存储使用次数。由于本程序注重缓存功能实现，在放入 ghost 表时，将次数进行置零操作（会浪费小部分存储空间），其存储结构是通过四个哈希链表完成的。如下表所示。

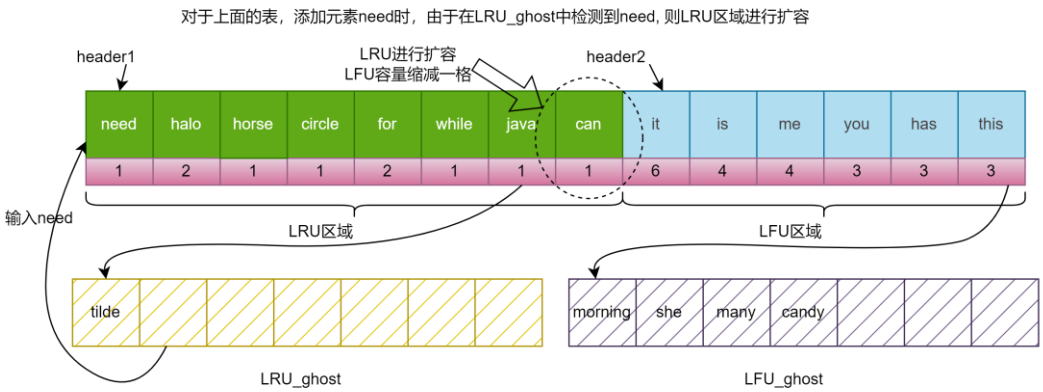


在输入过程中，对于任意一部分存储满时，如果继续加入元素,则插入对应存储部分的ghost 表中。

而对于在LRU_ghost 中的数据,储存在表内且在访问次数达到一定数量(transform time)时，数据会加入到 LFU 表中，转换为常访问的元素。



在上图里,每一次在输入数据时，先检查两个 ghost cache 中是否有对应的数据。如果数据在 LRU_ghost 中，说明目前访问元素的方法更接近于访问陌生元素。此时就会扩大 LRU 区域的空间并将这个元素放入 LRU 表内。而如果数据在 LFU_ghost 中出现，说明目前访问方法更多频繁访问某些元素，此时，我们让 LFU 区域的空间增加而让 LRU 区域减小，这样就实现了动态分配有限的缓存空间。



通过如上设计,我们就实现了有限内存的动态分配,在大量少重复访问不同元素时，LRU 区域长度会自适应增加，而在多次访问少量元素时,LFU 空间增加,这样就实现了动态分配。

三 详细设计

本程序着重于使用双向循环链表实现 ARC 算法中内存空间的动态分配问题。运行环境是 visual studio 2019 (需要 c++11 以上运行环境)

为了能够实现 ARC 缓存的功能， 定义两个类 LRUcache 和 LFUcache,由于每一个各自有一个缓存表空间 cache 和一个 ghost 表空间，分别在两个类中各自定义 cache_map 和 ghost_map,用来分别使用哈希表存储索引，进行高速搜索。另外对应有 cache 对应的 size 和 ghost_size 来检查表是否满或者未满进行动态删除元素。在判断表满时，size 对应的容量是 capacity，而 ghost_size 对应的容量是 ghost_capacity。

我们可以称使用哈希表存储链表功能的存储结构为哈希链表。哈希链表结合了哈希表查找快和链表的有序特点，使用哈希表存储链表的节点，可以实现时间复杂度为 O(1)的查找速度。使用哈希链表是缓存中一种很高效的存储办法。

然后在大的类 ARCcache 里面各包含一个这类，我们只需要在 ARCcache 的类中，让两个链表存储空间的总和相同即可,因此我们需要在 LRUcache 和 LFUcache 中分别实现如下的算法：

在添加元素时，首先搜索在 ghost 中是否出现，如果出现则调用其中一表的扩容和另一表的缩减函数, 如果没有则调用 put 函数

1. 向链表中添加元素的 put 函数，在表满情况下，则弹出(淘汰)表尾的元素并放入 cache 中，其中在向 ghost 添加元素时，如果 ghost 已满，则自动弹出尾部的元素。
2. 在 LRU 链表中元素使用达到一定次数时，需要返回一个值，提示 put 函数加入一个相同的元素到 LFU 链表中。
3. 扩容函数: 在表中插入元素同时判断目前是否能够扩容，如果能则让表容量+1
4. 缩减函数: 让表容量-1，如果表满的话需要另外移除表尾的元素。

因此分别 LRUcache 和 LFUcache 类本身，需要定义的函数有 put(),即在该部分插入元素。除此以外，还需要分别定义令其改变空间的函数 Add()和 Subtract(),分别对应加上一个元素空间和减去一个元素的空间。

另外里面定义的小函数比较多，往往都是在 put 里多次重复调用的，所以都做了封装，并分别

需要说明的是，在主函数中，定义 size 为 5，则分配空间是其 2 倍(10)， 即为 LRU,LFU 初始时分配相同的空间,另外为了简便，分配的 ghost 数组空间时

四 使用说明、测试分析及结果

1、 说明如何使用你编写的程序；

每一次输入一个词语按下回车，程序会自动统计词频并输出缓存的情况(附注:把 main 函数第 12 行的 true 改为 false 可以不输出 ghost 的缓存内容)

如果输入的是 exit,结束程序。

程序将类定义，结构体定义和函数实现分别放在多个 cpp 文件中，只需运行 main 即可。

2、 测试结果与分析；

程序目前经过一定量的测试，由于程序规模比较大，所以过多的测试没有进行，对于较为刁钻的数据输入可能以后出现报错情况，但是目前已经测试的数据输出结果均为正确结果。比较刁钻的数据比如设置缓存空间为 1，这个未经测试，目前已测试的部分使用的给定 size=5(即总空间为 10)，程序均能够输出正确结果。

对于本程序的性能，在 LRU 的插入,查找和删除的复杂度为 O(1)，而在 LFU 部分由于是有序插入,插入时间复杂度为 O(n),而查找和删除的复杂度为 O(1)， 对于 LFU 本身其实可以通过双哈希表的方法将其插入复杂度降低到 O(1)的，但是时间原因我没有进行实现。

相对实用的 ARC 缓存，ghost 数组中本程序连同 times 置零存储，浪费了少量存储空间。

另外，在本程序设计中，对于一个表(LFU 或者 LRU)全部被占用空后，插入数据时选择放弃插入的方式，这会遇到一些问题，比如 LFU 的表全部占满空间之后，可能出现数据无法插入的问题。比较好的方法是至少留一个存储空间或插入 ghost，不过时间原因也没有编写。

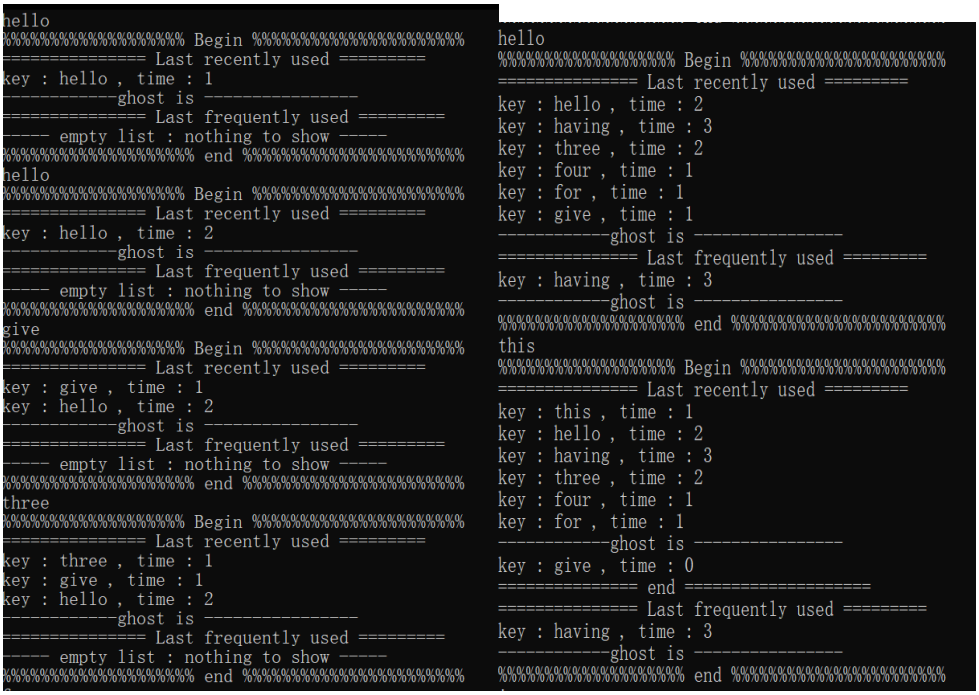
另外还有一些需要改进的点，比如有很多代码再 LFU 和 LRU 里面是重复的，如果能使用类的继承，可以精简不少代码(但是那个比较难，怕出其他问题我没敢做)

3、 调试过程中遇到的问题是如何解决提以及对设计与实现的回顾讨论和分析

遇到的困难主要是在整体布局方面的，由于程序较为繁杂，我首先实现了 LRC 算法，然后经过不少的思考进行了 ARC,LFU 和 LRU 类和方法的定义上。其中返回值也是问题，由于有可能修改头部，最初使用传入头部并返回对应的头部的方法来定义函数，但是由于可读性太差和参数传递繁琐（这样对应的表也需要传入）等，最终使用了定义一个 is_ghost 布尔参数判断，分别决定修改哪个头部的问题。

4、 运行界面

在下面的程序中，设置临界加入输入 hello 两次，右图是输入三次 hello 后，hello 加入到 LFU 缓存中的示例。



五、实验总结

程序的想法是 15 日中午进行选题的，下午进行整体布局，以及类定义和方法函数布局设计。另外花了不少时间在搜资料和弄懂过程上。

程序编写和调试完毕是 16 日下午，整个程序共用大约两天时间。

通过实验对缓存机制有了更为深刻的理解，也对当前数据结构在实际程序中的应用增长了一次有益的实践。

附注：由于网上这一方面的较为详细的讲解资料确实很缺乏，我会将此实验报告内容作为讲解发布文章到 CSDN 作为 ARC 算法讲解(用户名:程序菜鸟一只)

教师评语：

实验成绩：

指导教师签名：

批阅日期：