Assignment 4

Department of Informatics

Fall/Winter 2021- 2022

Professor: E. Markakis

Dionisios Rigatos (3200262)

PROJECT REPORT – LRU CACHE

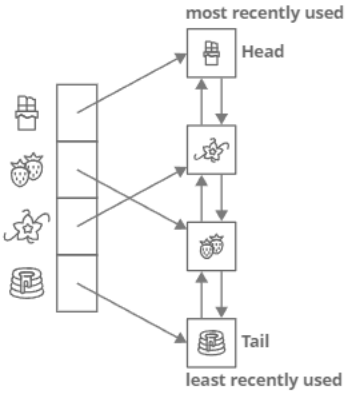
LRU Cache Implementation

The purpose of this assignment is to implement a quick LRU Cache in Java without utilizing any of Java's built-in data structures in the most time-efficient way possible.

For this implementation, a combination of a Doubly Linked List and a Hash Table with Linear Probing will be used. By combining these two data structures we will achieve constant time access to and removal of the least recently used element and almost constant access to any other element, which translates to fast updating of the elements in the cache. Each Hash Table element will be a pointer to a node in the Doubly Linked List, which will work as a priority queue with constant time eviction/insertion from/to any point (due to next/prev pointers in each DLL node) and constant time access to any element because of the Hash Table pointer. The main drawback of this implementation is the utilization of space is:

$$S(LRU\ Cache) = S(Doubly\ Linked\ List) + S(Hash\ Table) = n + 3n = 4n = O(n)$$

Where 3n is the product of multiplication of the cache capacity by the constant 3 to avoid clustering and to ensure a load factor of around 33%.



Storing Data

The main idea here is that while the cache has not reached its maximum capacity, data will be stored from the disk into the Hash Table and then pushed to the head of the Doubly Linked List (where the Most Recently Used element is located). If the element is already in the cache, its priority is updated by getting reinserted to the top of the Doubly Linked List. Finally, if the cache is full, the LRU element is evicted (removed from the Hash Table and popped from the tail of the DLL) and the newly inserted item is added to hash table and then to the head of the DLL.

In total, we have a worst case of O(n) insertion because of the Linear Probing schema. However, the worst case probability is very low due to the quality of the hashing function as well as the low load factor, which leaves us with an average insertion of Θ(1) (or 2.5 probes to be more specific). Removing elements from the Hash Table (in the store data in a full cache scenario) is slightly more costly as we look for the element in the table and if it's found, all of the other elements with the same hash value have to be moved to the left so as to make them available in future searches. This movement of data does not have an impact in the asymptotic analysis of our Hash Table or cache, and is O(n).

Accessing Data

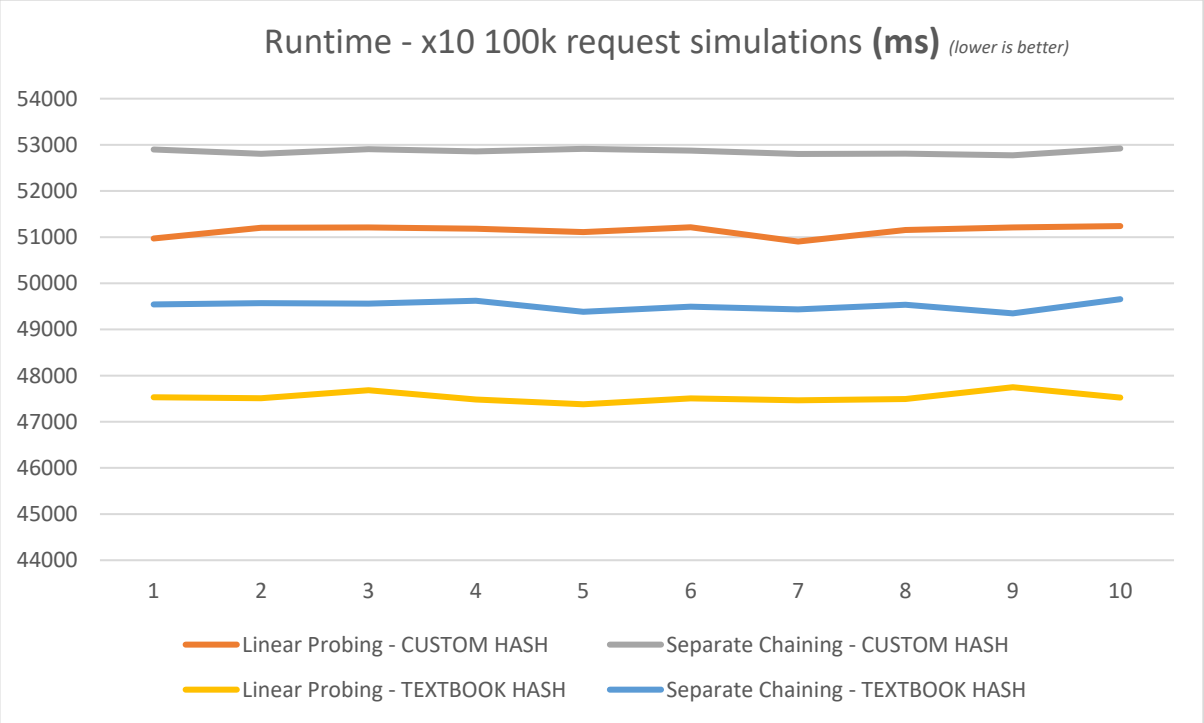
Similar to storing data, accessing an element has a worst case scenario of O(n), which is extremely improbable as the average-case analysis is Θ(1) (1.5 probe for a successful search or cache hit, 2.5 for an unsuccessful search or cache miss). If the search is unsuccessful, we simply return null. Otherwise, we return the element's value after its priority is updated by reinserting it into the DLL.

Hit Ratio, Misses etc.

Our LRU Cache Class maintains an eager approach on keeping track of lookups, hits and misses, which are incremented/decremented within the lookup operation for accessing data, allowing us to return the amount of cache hits, misses, lookups as well as the total hit ratio in O(1) time.

Linear Probing Schema

The use of Linear Probing as the Hash Table's schema is not random. It was compared against Separate Chaining using two different hash functions and was a clear winner in multiple simulations. The board below shows the running time for the dataset-5000.dat & requests-100000.dat files using a cache of size 500.



Hash Function

The Hash Function used in this implementation is converting the hashcode of a key to an array index by utilizing Java's hashCode() method, masking off the sign bit and computing the remainder when dividing by the capacity of the Table, as shown in the textbook Algorithms 4th Edition (Robert Sedgewick, Kevin Wayne) which is mentioned in the sources.

Bibliography & Sources:

[1] Sedgewick, R., Wayne, K. (2011). Algorithms, 4th Edition.. Addison-Wesley. ISBN: 978-0-321-57351-3

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.

[3] Markakis E. (2021/2022) INF231 Course. AUEB Department of Informatics.

[4] https://www.interviewcake.com/concept/java/lru-cache

[5] https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)