I. Introduction

Caching is fundamental to modern computing and has an array of applications such as in operating systems, databases, web servers, data compression etc. The design attempts to create a generic cache using randomization.

II. Data Structures

The underlying data structure for this cache is the *LinkedHashMap* with access order enabled.

III. Methodology

The implementation supports both the LRU and MRU as well as provides sub-classing support to allow a client to provide their own custom algorithm for cache eviction. However, this approach deviates from the conventional LRU and MRU. The conventional LRU relies on evicting the eldest member of the cache when the cache has reached its capacity in order to make new room for inserting a new member. In the case of MRU, the most recent member accessed is removed in order to make new room for inserting a new member. This approach applies the technique of randomization to both LRU and MRU for eviction. Randomization has very interesting implications in Computer Science and it has even been proposed to solve problems such as PageRank. Instead of choosing the eldest member (as in the case of LRU), this implementation instead selects a random member to evict based on the notion that if the keys of this cache could be represented as an Array, then the older members would be closer to the left-end of the array and the newer members would be to the right-end of the array. After selecting a pivot from this array, the keys to the left would be older members and the keys to the right would be newer members. To evict a member, a random index is selected from the left of this pivot array for LRU, and for MRU a random index is selected from the right of this pivot. Given that it cannot be predicted which member to evict on the next cache hit, it is best to leave eviction to the natural laws of randomization. It has been proposed that for very large caches randomized eviction works out better.

IV. Performance/Improvements

The *LinkedHashMap* uses an internal doubly-linked list which may slightly affect performance but it does have better performance during iteration.
A much better performing algorithm than conventional LRU is the Adaptive Replacement Cache Algorithm (ACU). It has been proven to show better performance.

V. How to run the code: The code can be cloned from: https://github.com/knasim/nway-cache There is also a zipped artifact in this public repo. The project uses gradle and can be imported into your IDE of choice. Test packages contain unit tests for the Cache and the Client.