1)

Η δομή που χρησιμοποιούμε είναι ένας συνδυασμός Doubly Linked List και HashMap τα οποία υλοποιούνται με την χρήση δεικτών head, tail, l, r και ενός πίνακα. Παράλληλα χρησιμοποιούμε και μια κλάση Record<K, V> η οποία περιέχει το key, value, left\_pointer και right\_pointer. Ουσιαστικά τοποθετούμε κάποιο αντικείμενο τύπου Record<K, V> στον πίνακα μέσω μιας hashing function και της χρήσης **γραμμικής διερεύνησης** (δεν είναι τυχαίο! Αναλύεται πιο κάτω το γιατί), κάνοντας τις απαραίτητες αλλαγές προκειμένου να διατηρούμε διαρκώς την σωστή σειρά προτεραιότητας των στοιχείων της cache.

2)

void shiftKeys(int currentPosition):

Η συνάρτηση αυτή φαίνεται με μια πρώτη ματιά περίεργη, ωστόσο ο στόχος της είναι πολύ απλός. Κάθε φορά που αφαιρούμε ένα entry από την cache χρησιμοποιούμε την συνάρτηση προκειμένου να φέρουμε το κάθε entry όσο πιο κοντά μπορούμε στην θέση που ορίζει η συνάρτηση hash με είσοδο το hashCode του. Για παράδειγμα αν έχουμε ένα αντικείμενο με hash = 6 και βρίσκεται στην θέση 9, τότε μετά από ένα remove **πιθανόν** να βρίσκεται στην θέση 8,7 ή 6 , με αποτέλεσμα η αναζήτηση να γίνεται πιο γρήγορα. Προφανώς η συνάρτηση δεν βελτιώνει όλες τις θέσεις και η βελτίωση δεν είναι πάντοτε σπουδαία, αλλά γενικά μας συμφέρει να την χρησιμοποιούμε.

**Time complexity: O(n) where n = totalCapacity**

void addEntry(int position):

Mέσω της συνάρτησης αυτής απλώς αλλάζουμε τις τιμές μερικών pointers, ώστε η νέα είσοδος να είναι στο τέλος του Doubly Linked List, δηλαδή να έχει την μικρότερη προτεραιότητα εξόδου από την cache.

**Time complexity: O(1)**

void removeEntry(int position):

Mέσω της συνάρτησης αυτής απλώς αλλάζουμε τις τιμές μερικών pointers, ώστε να αφαιρέσουμε πρακτικά από το Doubly Linked List το στοιχείο στο head, καθώς και να φέρουμε ένα νέο στοιχείο στην θέση αυτή .

**Time complexity: O(1)**

double getHitRatio(), long getHits(), long getMisses(), long getNumberOfLookups():

Απλές συναρτήσεις που κάνουν ό,τι λέει το όνομά τους.

**Time complexity: O(1) για όλες**