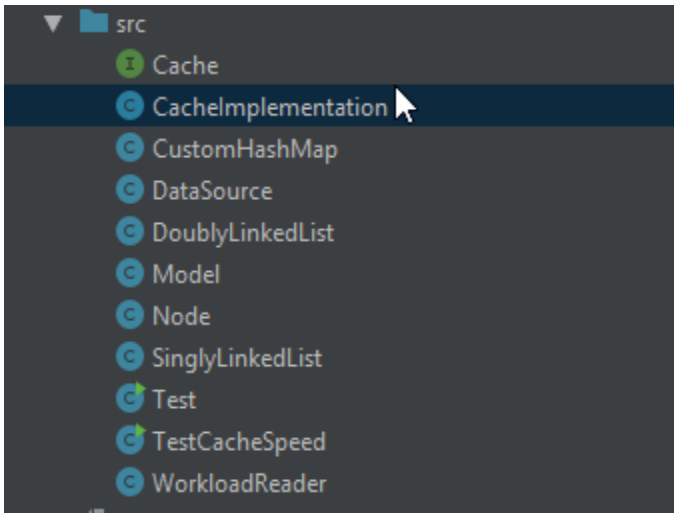
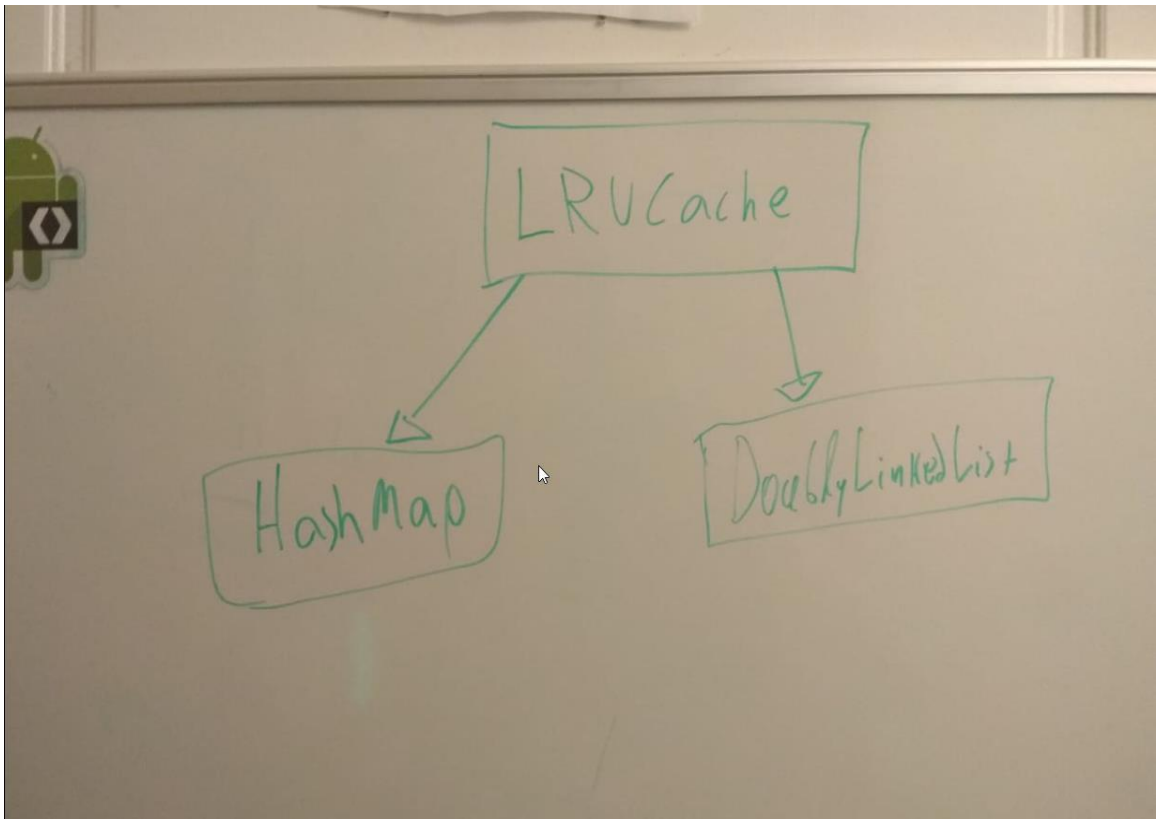
4η ΕΡΓΑΣΙΑ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Α ΜΕΡΟΣ





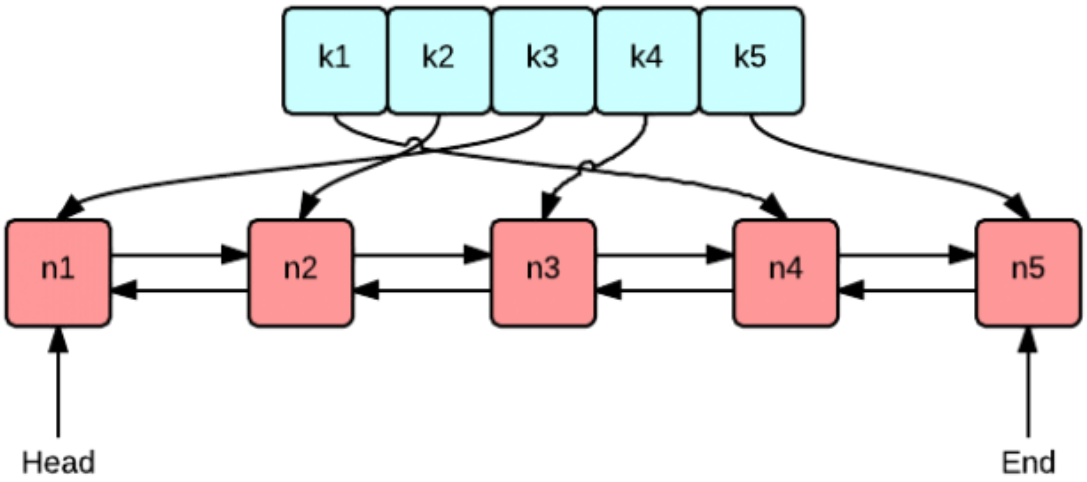
Όπως φαίνεται και στην πρώτη εικόνα με τα αρχεία που υλοποιήσαμε έχουμε χρησιμοποιήσει τις εξής δομές:

1. **HashMap(αλυσιδωτή)**
2. **SinglyLinkedList**
3. **DoublyLinkedList**

Αρχικός στόχος ήταν να φτιάξουμε ένα loosely coupled και robust source code το οποίο να ακολουθεί καλές τεχνικές προγραμματισμού (aka SOLID principles). Στην συνέχεια με την βοήθεια των γνώσεων που λάβαμε από το μάθημα καταφέραμε να βελτιώσουμε την ταχύτητα/απόδοση του προγράμματος.

Η λειτουργεία της δομής(cache) είναι πολύ απλή. Έχουμε ένα hashMap που αποτελείται από SinglyLinkedList και έχουμε και ένα doublyLinkedList. Με την βοήθεια αυτών είναι δυνατή η σωστή λειτουργία των δύο βασικών λειτουργειών της cache, της αποθήκευσης και της αναζήτησης. Στην ουσία κάθε φορά παίρνουμε μια απαίτηση από τον χρήστη και ψάχνουμε αν η απαίτηση αυτή(το στοιχείο) βρίσκεται στην cache άρα αν υπάρχει στο hashMap. Αν υπάρχει τότε με βάση το κλειδί του αντικειμένου(κάνουμε hashing) πάμε στην κατάλληλη θέση στον πίνακα με τα singlyLinkedList της HashMap και ψάχνουμε αν υπάρχει, αν ναι τότε όλα καλά, ενημερώνουμε κατάλληλα την doublyLinkedList ώστε το στοιχείο αυτό να πάει στην κορυφή της αφού αναζητήθηκε πιο πρόσφατα απο όλα(έχουμε LRU cache) και πάμε στην επόμενη απαίτηση του χρήστη. Αν όμως το στοιχείο αυτό δεν έχει εισαχθεί τότε ψάχνουμε αν είναι από τα στοιχεία που υπάρχουν στο data μας και ένα ναι τότε πάμε και το αποθηκεύουμε στην cache, άρα στο HashMap και στο doublyLinkedList. Η αποθήκευση είναι απλούστατη. Αρχικά διαγράφουμε από το HashMap το στοιχείο που έχει παραμείνει παραπάνω στην cache και αυτό είναι το tail του doublyLinkedList. Αυτό το κάνουμε ώστε τα στοιχεία του hashMap να μην περνάνε σε πλήθος το cache size. Στη συνέχεια τοποθετούμε το στοιχείο προς εισαγωγή στην κατάλληλη λίστα(θέση) του πίνακα με singlyLinkedList που περιέχει η hashMap. Η τοποθέτηση γίνεται με βάση το hashingKey. Έπειτα τοποθετούμε το στοιχείο αυτό και στην doublyLinkedList και επειδή εισάχθηκε μόλις τώρα μπαίνει στο head(θέλουμε να κρατήσουμε την LRU ιδιότητα της cache). Μετά συνεχίζουμε με το επόμενο request(απαίτηση) του χρήστη.

Στην παρακάτω εικόνα φαίνεται πως σχετίζονται οι δομές μεταξύ τους και άρα το πώς επιτυγχάνεται η υλοποίηση μιας cache.



Β ΜΕΡΟΣ

Ο λόγος που χρησιμοποιήσαμε τα συγκεκριμένα data structure είναι διότι όπως διαβάσαμε στο internet(stackOverflow) αυτές οι δομές προσφέρουν ταχύτατη cache που είναι και ο σκοπός της εργασίας. Στην ουσία το hashMap(που έχει SinglyLinkedList) χρησιμοποιείται επειδή θέλουμε να έχουμε γρήγορη εισαγωγή, αναζήτηση και διαγραφή και χάρη στο hashing οι λειτουργείες αυτές πραγματοποιούνται σε Θ(1). Η doublyLinkedList προσφέρει την ιδιότητα LRU και χρειάζεται ώστε οι λειτουργεία διαγραφής από το hashMap να γίνεται σε Θ(1). Και επειδή η doubyLinkedList έχει στο τέλος το lru στοιχείο αυτό μας επιτρέπει άμεση πρόσβαση και διαγραφή του από το hashMap, άρα και διαγραφή από την cache. Επομένως γίνεται φανερό ότι χρησιμοποιήσαμε τις παραπάνω δομές για λόγους ταχύτητας.

Το υπολογιστικό κόστος:

Η **πολυπλοκότητα της μνήμης** είναι O(N) καθώς είναι αντίστοιχη με το cache size δηλαδή με το μέγιστο πλήθος στοιχείων που μπορεί να περιέχει η cache.

Τώρα ας δούμε την πολυπλοκότητα των δυο βασικών λειτουργειών της cache, της LookUp() και Store().

Η **LookUp()** έχει πολυπλοκότητα Θ(1), O(N), όπου N το cache size. Έχει χειρότερη πολυπλοκότητα O(N) διότι αρχικά πάει στο hashMap και ψάχνει αν υπάρχει το στοιχείο, δλδ καλεί την find της hashMap. Η hashMap επειδή

αποτελείται από ένα πίνακα με SinglylinkedList με τη σηρά της καλεί την find της linked list και επειδή έχουμε Ν στοιχεία η χειρότερη περίπτωση είναι όλα τα στοιχεία να μπούνε σε μια συγκεκριμένη θέση του πίνακα της hashMap δλδ το ίδιο linkedList οπότε θα ψάξουμε το πολύ Ν στοιχεία. Στη συνέχεια και εάν το στοιχείο βρέθηκε πρέπει να ενημερωθεί και η θέση του στην DoublyLinkedList και να πάει στην κορυφή σαν το πιο πρόσφατα αντικείμενο που αναζητήθηκε. Η lookup καλεί την moveNodeAtHead της doublyLinkedlist , η οποία έχει πολυπλοκότητα O(1). Ωστόσο επειδή στο hashMap για να εισάγουμε τα στοιχεία χρησιμοποιούμε hashing παρατηρούμε ότι τα στοιχεία κατανέμονται στην hashMap με τρόπο τέτοιο ώστε η λειτουργεία της LookUp() να γίνεται σε Θ(1).

Η **STORE()** έχει πολυπλοκότητα Θ(1), O(N), όπου Ν το cache size.Στην ουσία επειδή πάμε να εισάγουμε ένα νέο στοιχείο τα στοιχεία στην cache δεν πρέπει να περνάνε το cache size και για αυτόν τον λόγο βρίσουμε το lru στοιχείο από το doublyLinkedList σε χρονο O(1) και στη συνέχεια το διαγράφουμε από το hashMap με την μέθοδο delete λειτουργεια O(N) αλλα Θ(1). Έπειτα το στοιχείο προς εισαγωγή τοποθετείται στο hashMap με την μέθοδο putNewNode της HashMap η οποία με τη σειρά της καλεί την put της singlyLinkedList, πολυπλοκότητα O(N) και Θ(1) για τους ίδιους λόγους που ή find της hasHMap έχει πολυπλοκότητα O(N) και Θ(1).

Γ ΜΕΡΟΣ

Πηγές:

- https://medium.com/@krishankantsinghal/my-first-blog-on-medium-583159139237
- https://www.geeksforgeeks.org/lru-cache-implementation/
- https://stackoverflow.com/questions/2504178/lru-cache-design

.....

ΤΖΕΝΗ ΜΠΟΛΕΝΑ 3170117

ΚΩΝΣΤΑΝΤΙΝΟΣ ΝΙΚΟΛΟΥΤΣΟΣ 3170122

.....