

# DomesDedomenwn

Το πρόγραμμα DomesDedomenwn προσομοιώνει την λειτουργία μιας μνήμης cache σε Java.

Ο σκοπός αυτής της προσομοίωσης είναι η μνήμη cache να έχει ένα σταθερό (σχετικά μικρό) μέγεθος και να δέχεται αντικείμενα. Όταν γεμίσει η μνήμη πρέπει να βγάζει ένα στοιχείο έξω από την μνήμη για να κάνει χώρο για ένα επόμενο. Το κριτήριο που καθορίζει το ποιος θα βγει για να κάνει χώρο αποφασίζεται με τον τύπο της cache (LRU, MRU, LFU). Ο τύπος της cache ορίζει το τι replacement priority έχει η συγκεκριμένη μνήμη.

## Δουλειά του προγράμματος είναι:

- 1) να υλοποιεί τους τρεις τύπους μνήμης cache,
- 2) να κάνει τα κατάλληλα test για κάθε μνήμη
- 3) και να εμφανίζει τα αποτελέσματα της λειτουργίας hit/miss.

## 1. Τύποι Μνήμης Cache

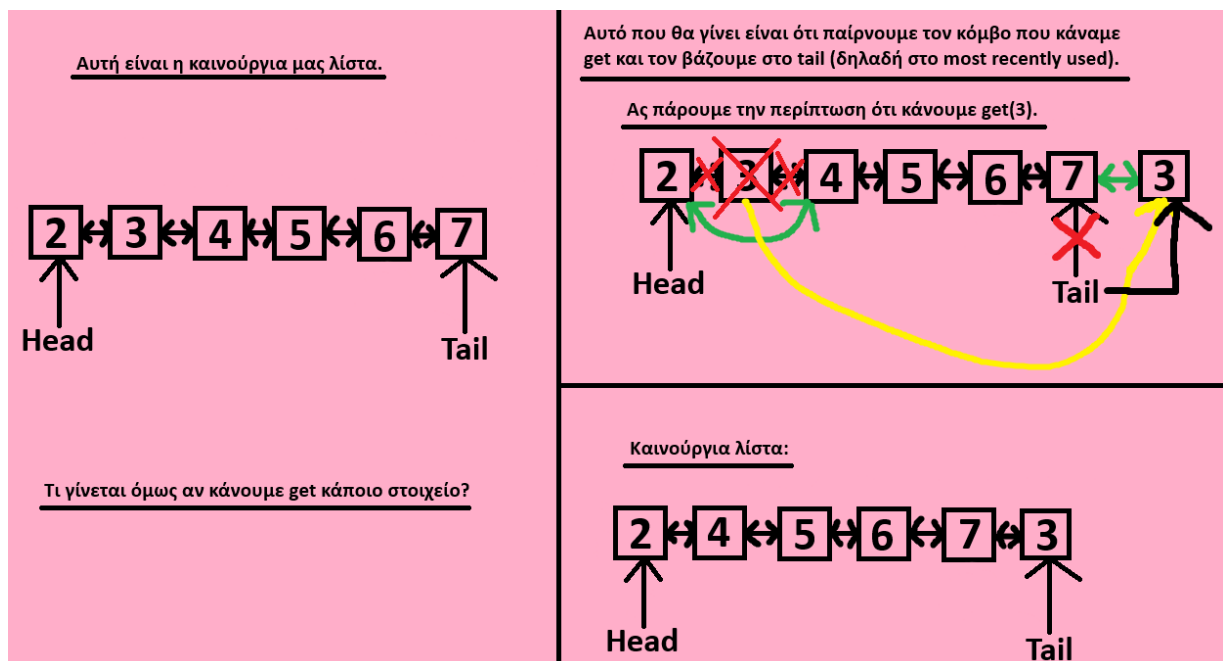
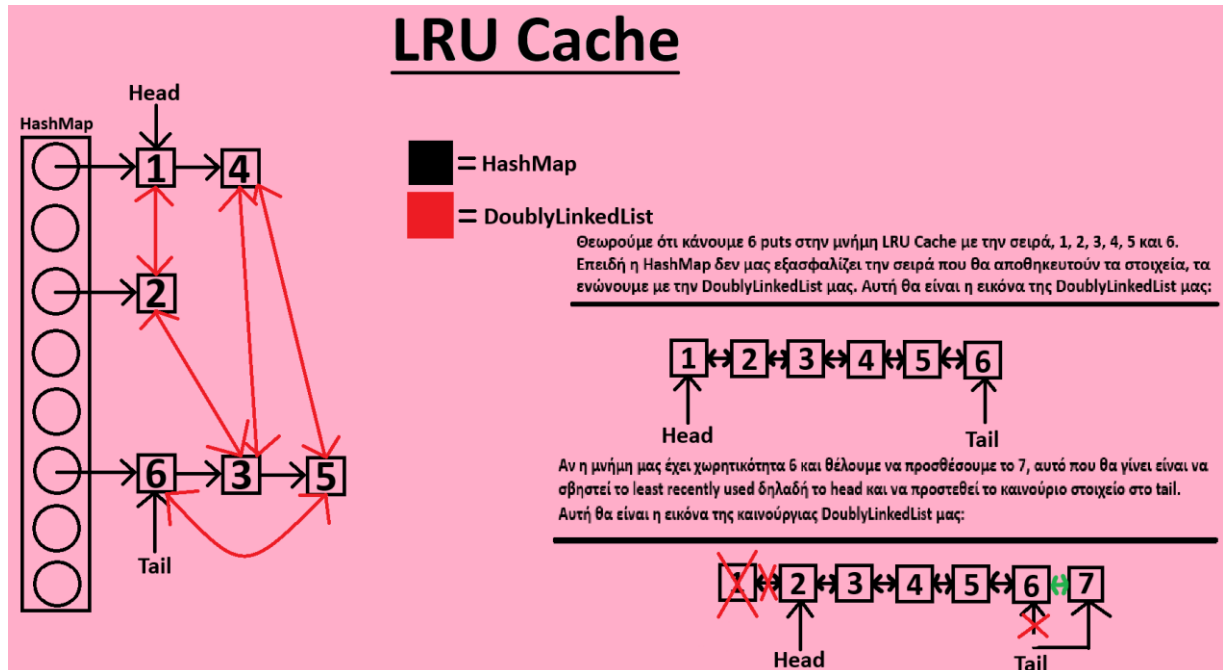
### Τύπος 1 (LRU Cache):

Ο πρώτος τύπος μνήμης cache είναι η LRU (least recently used) Cache που το αντικείμενο που θα βγει για να κάνει χώρο είναι αυτό που χρησιμοποιήθηκε λιγότερο πρόσφατα.

Ο τρόπος υλοποίησης της LRU θα γίνει με την βοήθεια της έτοιμης υλοποίησης πίνακα κατακερματισμού της Java το HashMap και μια δικιά μας υλοποίηση μιας διπλά συνδεδεμένης λίστας με βοήθεια της εσωτερικής κλάσης Node. Η σύνδεση μεταξύ των δύο είναι εφικτή επειδή η διπλά συνδεδεμένη λίστα κουβαλάει το key και το value που χρειαζόμαστε για το αντικείμενο και το HashMap κουβαλάει σαν key το ίδιο key με το Node και σαν value δέχεται το ίδιο το Node. Έτσι για την αναζήτηση ενός κόμβου με το key χρησιμοποιούμε το HashMap και μας επιστρέφει το value το που είναι το ίδιο το Node οπότε η εύρεση κάποιου κόμβου με ένα key γίνεται σε  $O(1)$  χρόνο. Συνεπώς,  $O(1)$  είναι και η διαγραφή γιατί αφού με την αναζήτηση (που είναι  $O(1)$ ) παίρνουμε τον κόμβο που θέλουμε να διαγράψουμε, το μόνο που μένει να κάνουμε είναι να αλλάξουμε το head και να πειράξουμε τα κατάλληλα next και prev για να προσαρμόσουμε την λίστα μας που επίσης είναι  $O(1)$ . Κάθε διαγραφή γίνεται από την αρχή της λίστας (δηλαδή το head) κάθε φορά που γεμίζει η μνήμη cache. Τέλος, και η πρόσθεση στοιχείου είναι  $O(1)$  γιατί γίνεται από το τέλος και το μόνο που κάνουμε είναι να πειράξουμε το tail με τον κόμβο που θέλουμε να βάλουμε (και το next και prev).

Συμπερασματικά, κάθε λειτουργία της LRU Cache γίνεται σε  $O(1)$  χρόνο.

## Σχήμα για LRU Cache:



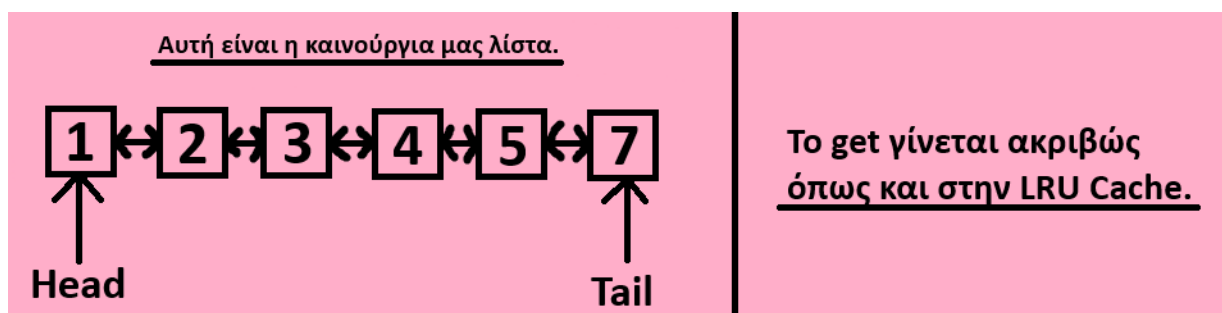
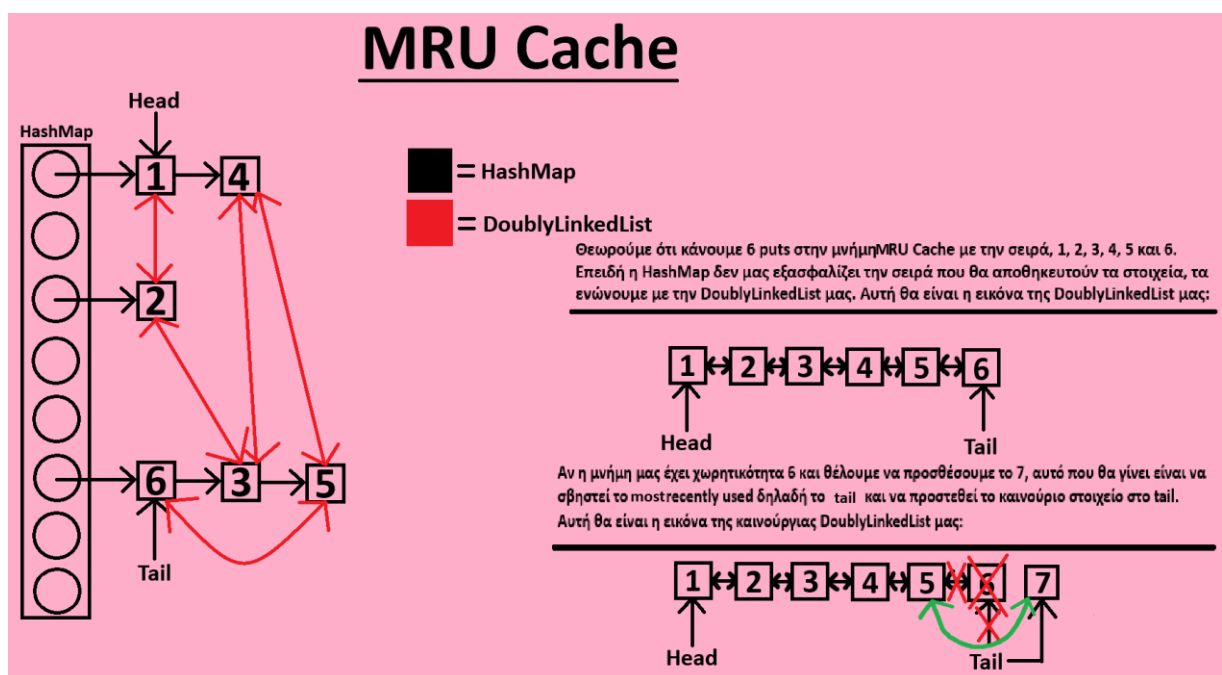
## Τύπος 2 (MRU Cache):

Ο δεύτερος τύπος μνήμης cache είναι η MRU (most recently used) Cache που το αντικείμενος που θα βγει για να κάνει χώρο είναι αυτό που χρησιμοποιήθηκε πιο πρόσφατα.

Ο τρόπος υλοποίησης της MRU θα γίνει ακριβώς όπως έγινε στην LRU με την μόνη διαφορά ότι κάθε διαγραφή θα γίνει στο τέλος της λίστας αντί για την αρχή, που πάλι γίνεται σε  $O(1)$  χρόνο γιατί το μόνο που πρέπει να κάνουμε είναι να αλλάξουμε το tail και να πειράξουμε τα κατάλληλα next και prev για να προσαρμόσουμε την λίστα μας που επίσης είναι  $O(1)$ . Άρα κάθε διαγραφή γίνεται από το τέλος της λίστας (δηλαδή το tail) κάθε φορά που γεμίζει η μνήμη cache.

Συμπερασματικά, αφού η λειτουργία που αλλάξαμε στην MRU είναι  $O(1)$  και κάθε λειτουργία της LRU που χρησιμοποιήσαμε στην MRU είναι και αυτή  $O(1)$  (απόδειξη στον τύπο 1), κάθε λειτουργία της MRU Cache γίνεται σε  $O(1)$  χρόνο.

## Σχήμα για MRU Cache:



### Τύπος 3 (LFU Cache):

Work in progress...

## 2. Tests

Τώρα που εξηγήσαμε την υλοποίηση κάθε τύπου μνήμης cache, πάμε να δούμε τα tests που πρέπει να κάνει κάθε μνήμη για να θεωρηθεί σωστά υλοποιημένη.

Υπάρχουν τρία είδη tests που πρέπει να κάνουμε για κάθε μνήμη cache.

**Test 1 (general test):** Ελέγχει την ορθότητα της υλοποίησης της μνήμης cache.

**Test 2 (edge cases test):** Ελέγχει τις ακριανές τιμές της μνήμης cache.

**Test 3 (stress test):** Ελέγχει την ανθεκτικότητα της μνήμης cache.

Αυτά τα 3 test γίνονται για κάθε τύπο μνήμης cache από μία φορά.

Τώρα θα δείξουμε πως υλοποιούνται αυτά τα tests.

**Υλοποίηση test 1 (general test):** Ο τρόπος που το general test ελέγχει την ορθότητα της υλοποίησης της μνήμης cache είναι βάζοντας στην cache διπλάσιο πλήθος στοιχεία από αυτά που χωράει και ελέγχουμε αν όλα τα στοιχεία μπήκαν σωστά στην cache.

**Υλοποίηση test 2 (edge cases test):** Ο τρόπος που το edge cases test ελέγχει τις ακριανές τιμές της μνήμης cache είναι γεμίζοντας την και στην συνέχεια ελέγχει για κάθε φορά που βάζουμε καινούργιο στοιχείο αν μπήκε σωστά και αν διαγράφηκε αυτό που έπρεπε να διαγραφεί ανάλογα με τον τύπο της cache.

**Υλοποίηση test 3 (stress test):** Ο τρόπος που το stress test ελέγχει την ανθεκτικότητα της μνήμης cache είναι φορτώνοντας την μνήμη με έναν πάρα πολύ μεγάλο πλήθος στοιχείων και στην συνέχεια ελέγχοντας αν η μνήμη δεν έσκασε και επίσης αν τα στοιχεία που έμειναν στην μνήμη είναι αυτά που περιμέναμε ανάλογα τον τύπο της cache.

### 3. Λειτουργία Hit/Miss

Η σκοπός της λειτουργίας hit/miss είναι να εμφανίσει πόσες φορές η μνήμη cache περιέχει το στοιχείο που αναζητήσαμε (hit) και πόσες φορές δεν το βρήκε (miss) καθώς και το ποσοστό των hit και miss. Το πόσα στοιχεία θα αναζητήσουμε το καθορίζει η σταθερά OPERATIONS που και αυτή εμφανίζεται στα αποτελέσματα. Αυτή η διαδικασία γίνεται μια φορά για κάθε μνήμη.

#### Παράδειγμα αποτελέσματος λειτουργίας hit/miss για μία μνήμη:

**Total operations: 100000**  
**Cache Hits: 52300**  
**Cache Misses: 47700**  
**Hit Rate: 52,30%**  
**Miss Rate: 47,70%**

#### Το στοιχείο που αναζητάμε κάθε φορά είναι τυχαίο και ο τρόπος που διαλέγεται είναι ο εξής:

Έχουμε ένα σύνολο ακεραίων από 0 μέχρι 99999 που θα ονομάσουμε U. Χωρίζουμε το σύνολο U σε δύο ίσα υποσύνολα A από 0 μέχρι 49999 και B από 50000 μέχρι 99999. Θέλουμε να έχουμε κατανομή 80/20 στα υποσύνολα που σημαίνει ότι θέλουμε να υπάρχει 80% πιθανότητα να πάρουμε έναν τυχαίο αριθμό από το υποσύνολο A και 20% πιθανότητα να πάρουμε έναν τυχαίο αριθμό από το υποσύνολο B. Ο τρόπος που θα το κάνουμε αυτό είναι με την βοήθεια της Random που μας δίνει έναν τυχαίο αριθμό από το 0 μέχρι το range που του δώσουμε. Για την κατανομή 80/20 θα ζητήσουμε έναν τυχαίο αριθμό από 0 μέχρι 4. Θεωρούμε ότι η τιμή που ψάχνουμε είναι το 0. Μαθηματικά μιλώντας, έχουμε 20% πιθανότητα να πετύχουμε την τιμή που ψάχναμε (δηλαδή το 0) και 80% πιθανότητα να μην την πετύχουμε. Με αυτά τα δεδομένα, αν κάθε φορά που μας τυχαίνει το 0 παίρνουμε έναν τυχαίο αριθμό από το υποσύνολο B και κάθε φορά που δεν μας τυχαίνει παίρνουμε έναν τυχαίο αριθμό από το υποσύνολο A, έχουμε φτιάξει την κατανομή 80/20 όπου υπάρχει 80% πιθανότητα να πάρουμε έναν τυχαίο αριθμό από το υποσύνολο A και 20% πιθανότητα να πάρουμε έναν τυχαίο αριθμό από το υποσύνολο B.

Κατανομή 80/20 σε σχήμα:

