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 μνήμα 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

Αυτό το πρόγραμμα υποστηρίζει 3 τύπους μνήμης cache:

1) LRU Cache
2) MRU Cache
3) FRU Cache

Επειδή χρειάζονται διαφορετικές μεταβλητές για την υλοποίηση του LRU και του MRU από την υλοποίηση του LFU, έχουμε φτιάξει δύο εσωτερικές κλάσεις, την CachingLruMru και την CachingLfu, όπου και οι δύο είναι υποκλάσεις της εσωτερικής διεπαφής Caching. Η δουλειά της CachingLruMru είναι να υλοποιεί το get και το put για τους τύπους LRU και MRU και η δουλειά της CachingLfu είναι να υλοποιεί το get και το put για τον τύπο LFU. Όταν καλείτε η get ή η put σε μία μνήμη cache, αυτή καλεί αυτόματα την get ή την put της CachingLruMru ή της CachingLfu ανάλογα με τον τύπο της.

Τύπος 1 (LRU Cache):

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

Υλοποίηση και Order λειτουργιών της LRU Cache:

Ο τρόπος υλοποίησης της LRU θα γίνει με την βοήθεια της έτοιμης υλοποίησης κατακερματισμένης αποθήκευσης 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) χρόνου είναι και η διαγραφή γιατί αφού με την αναζήτηση (που είναι O(1) χρόνου) 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) χρόνου. 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) χρόνου 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 γίνετε σε O(1) χρόνο.

Λειτουργίες της LRU Cache:

get: Δέχεται ένα κλειδί, τοποθετεί τον κόμβο με αυτό το κλειδί στο τέλος της λίστας (στο most recent), ανεβάζει το hitCount κατά 1 και επιστρέφει την τιμή του κλειδιού. Αν το κλειδί δεν υπάρχει στην μνήμη, ανεβάζει το missCount κατά 1 και επιστρέφει null.

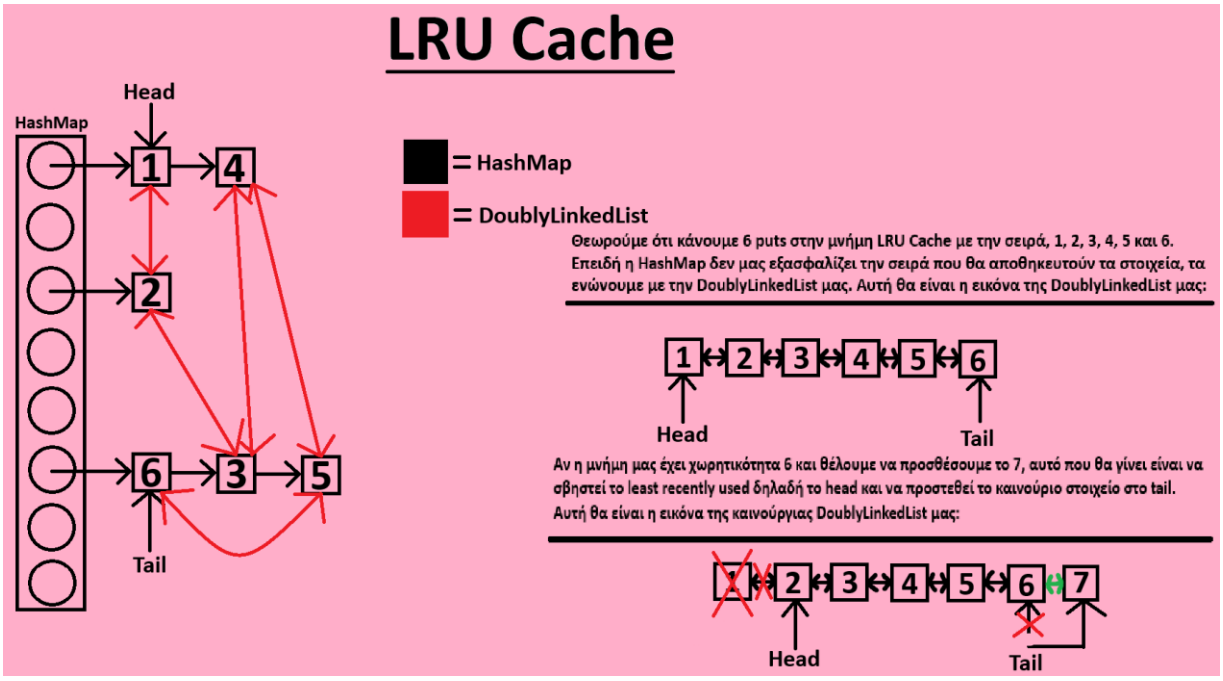
put: Δέχεται ένα κλειδί και μια τιμή, φτιάχνει έναν κόμβο με αυτό το κλειδί και τιμή, τον τοποθετεί στο τέλος της (στο most recent) και τοποθετεί το κλειδί και τον κόμβο στην μνήμη. Αν το κλειδί υπάρχει ήδη στην μνήμη, ανανεώνει στον κόμβο με αυτό το κλειδί την παλιά τιμή με την καινούρια και τον τοποθετεί στο τέλος της λίστας (στο most recent).

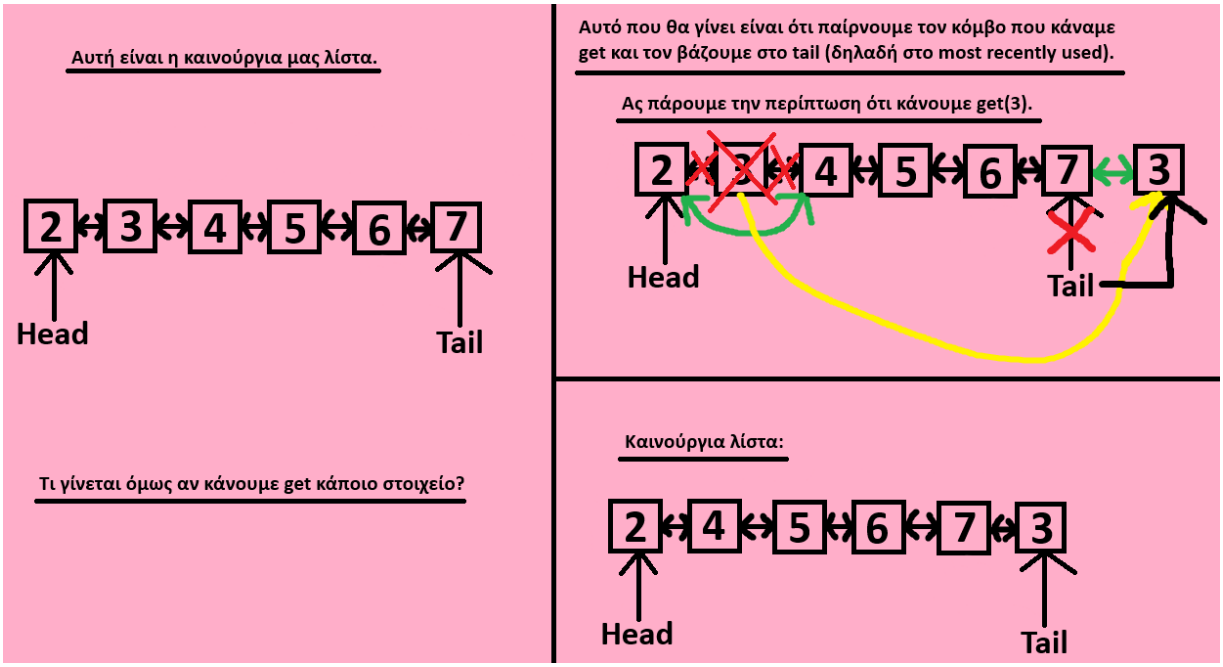
put (σε γεμάτη μνήμη): Στην περίπτωση που γίνει put σε γεμάτη μνήμη, πριν την εισαγωγή του καινούριου κόμβου διαγράφεται από την μνήμη ο πρώτος κόμβος της λίστας (το head που είναι το least recent) για να κάνει χώρο για τον καινούριο κόμβο και στην συνέχεια ξεκινάει η λειτουργία put.

getHitCount: Επιστρέφει το πλήθος των αναζητήσεων που υπήρχε το στοιχείο που ψάχτηκε στην μνήμη.

getMissCount: Επιστρέφει το πλήθος των αναζητήσεων που δεν υπήρχε το στοιχείο που ψάχτηκε στην μνήμη.

Σχήμα της LRU Cache:





Τύπος 2 (MRU Cache):

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

Υλοποίηση και Order λειτουργιών της MRU Cache:

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

Λειτουργίες της MRU Cache:

get: Δέχεται ένα κλειδί, τοποθετεί τον κόμβο με αυτό το κλειδί στο τέλος της λίστας (στο most recent), ανεβάζει το hitCount κατά 1 και επιστρέφει την τιμή του κλειδιού. Αν το κλειδί δεν υπάρχει στην μνήμη, ανεβάζει το missCount κατά 1 και επιστρέφει null.

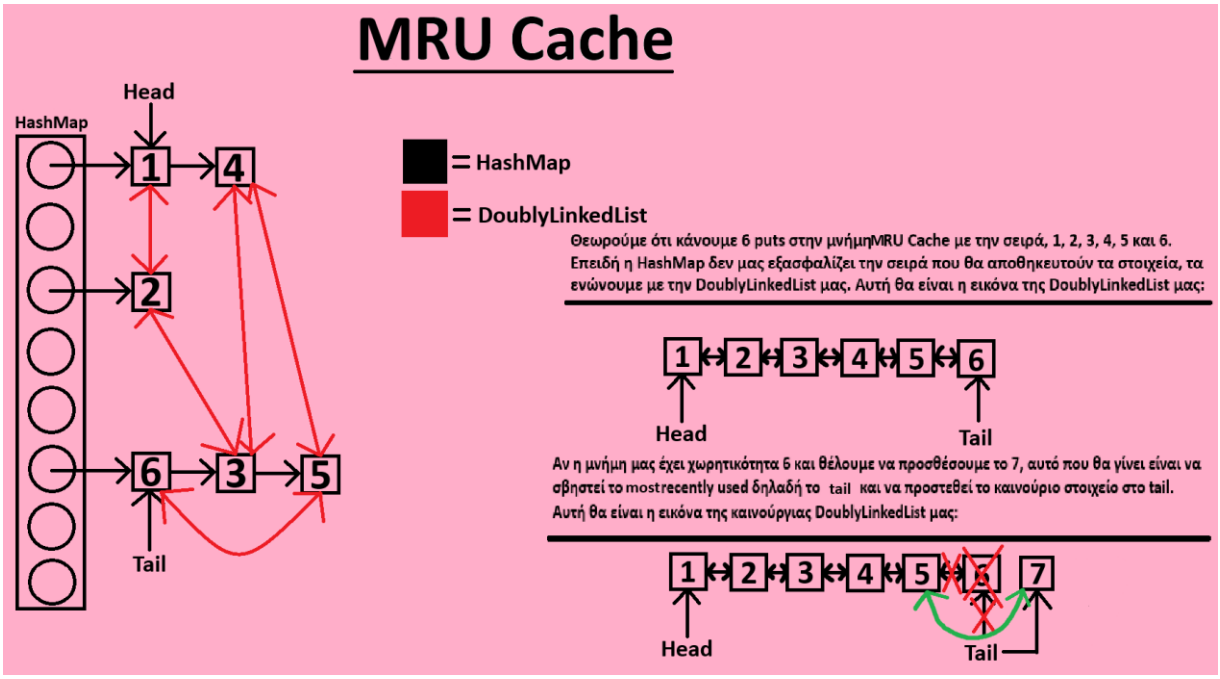
put: Δέχεται ένα κλειδί και μια τιμή, φτιάχνει έναν κόμβο με αυτό το κλειδί και τιμή, τον τοποθετεί στο τέλος της (στο most recent) και τοποθετεί το κλειδί και τον κόμβο στην μνήμη. Αν το κλειδί υπάρχει ήδη στην μνήμη, ανανεώνει στον κόμβο με αυτό το κλειδί την παλιά τιμή με την καινούρια και τον τοποθετεί στο τέλος της λίστας (στο most recent).

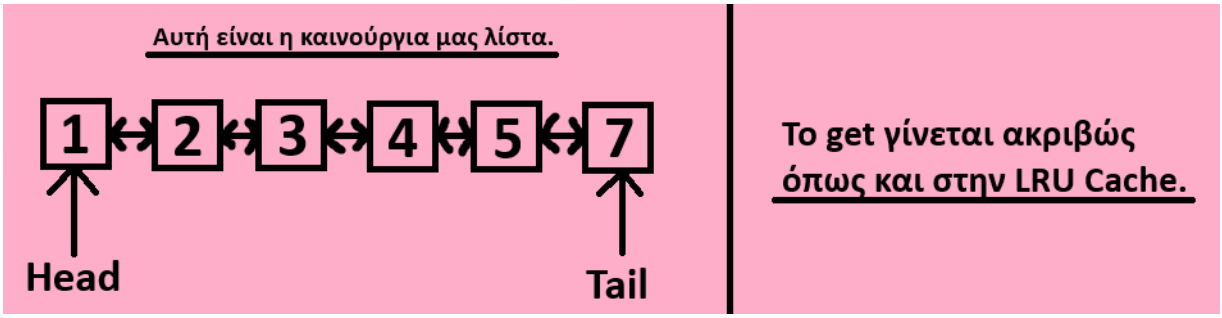
put (σε γεμάτη μνήμη): Στην περίπτωση που γίνει put σε γεμάτη μνήμη, πριν την εισαγωγή του καινούριου κόμβου διαγράφεται από την μνήμη ο τελευταίος κόμβος της λίστας (το tail που είναι το most recent) για να κάνει χώρο για τον καινούριο κόμβο και μετά ξεκινάει η λειτουργία put.

getHitCount: Επιστρέφει το πλήθος των αναζητήσεων που υπήρχε το στοιχείο που ψάχτηκε στην μνήμη.

getMissCount: Επιστρέφει το πλήθος των αναζητήσεων που δεν υπήρχε το στοιχείο που ψάχτηκε στην μνήμη.

Σχήμα της MRU Cache:





Τύπος 3 (LFU Cache):

Ο τρίτος τύπος μνήμης cache είναι η LFU (least frequently used) Cache που το στοιχείο που θα βγει από την μνήμη για να κάνει χώρο για το επόμενο είναι αυτό που χρησιμοποιήθηκε λιγότερο σιχνά.

Υλοποίηση και Order λειτουργιών της LFU Cache:

Ο τρόπος υλοποίησης της LFU θα γίνει με την βοήθεια των έτοιμων υλοποιήσεων δέντρου αναζήτησης δυαδικών κλειδιών και κατακερματισμένης αποθήκευσης της Java, το TreeMap και το HashMap. Το HashMap κρατάει σαν κλειδί το κλειδί που θέλουμε να αποθηκεύσουμε και σαν τιμή κρατάει την συχνότητα που έχει αναζητηθεί αυτό το κλειδί. Το TreeMap κρατάει σαν κλειδί μια συχνότητα και σαν τιμή ένα HashMap (με κλειδί και τιμή, το κλειδί και την τιμή που θέλουμε να αποθηκεύσουμε) που κρατάει όλα τα αποθηκεύσουμε) που κρατάει όλα τα αποθηκευμένα στοιχεία με αυτήν την συχνότητα. Για την αναζήτηση κάποιου στοιχείου δίνουμε το key που ψάχνουμε στο HashMap και αυτό μας επιστρέφει το frequency αυτού του κλειδιού. Παίρνουμε το frequency και το δίνουμε στο TreeMap και αυτό μας επιστρέφει το HashMap με τα στοιχεία που έχουν αυτό το frequency. Στο HashMap με τα στοιχεία που μας έδωσε το TreeMap δίνουμε το key (που θέλουμε την τιμή του) και μας επιστρέφει το value του στοιχείου που θέλαμε. Άρα για την αναζήτηση κάναμε 2 gets σε HashMap, που είναι O(1) χρόνου και 1 get σε TreeMap, που είναι O(log n) χρόνου, άρα η αναζήτηση είναι O(log n) χρόνου. Συνεπώς, O(log n) χρόνου είναι και η διαγραφή γιατί η διαγραφή γίνεται κάνοντας αναζήτηση (που είναι O(log n) χρόνου) για να βρούμε το HashMap με τα στοιχεία και κάνοντας remove στα 2 HashMap (που είναι O(1) χρόνου). Κάθε διαγραφή γίνεται από ένα στοιχείο τον HashMap που βρίσκεται σαν τιμή στο χαμηλότερο κλειδί του TreeMap (δηλαδή το least frequently used) κάθε φορά που γεμίζει η μνήμη cache. Τέλος, και η πρόσθεση στοιχείου είναι O(log n) χρόνου γιατί γίνεται κάνοντας αναζήτηση για να βρεθεί το HashMap που θα προστεθεί το καινούργιο στοιχείο (που είναι O(log n) χρόνου) και κάνοντας put στα 2 HashMap (που είναι O(1) χρόνου).

Συμπερασματικά, κάθε λειτουργία της LFU Cache γίνεται σε O(log n) χρόνο.

Λειτουργίες της LFU Cache:

get: Δέχεται ένα κλειδί, ανεβάζει κατά 1 την συχνότητα αυτού του κλειδιού, ανεβάζει το hitCount κατά 1 και επιστρέφει την τιμή του κλειδιού. Αν το κλειδί δεν υπάρχει στην μνήμη, ανεβάζει το missCount κατά 1 και επιστρέφει null.

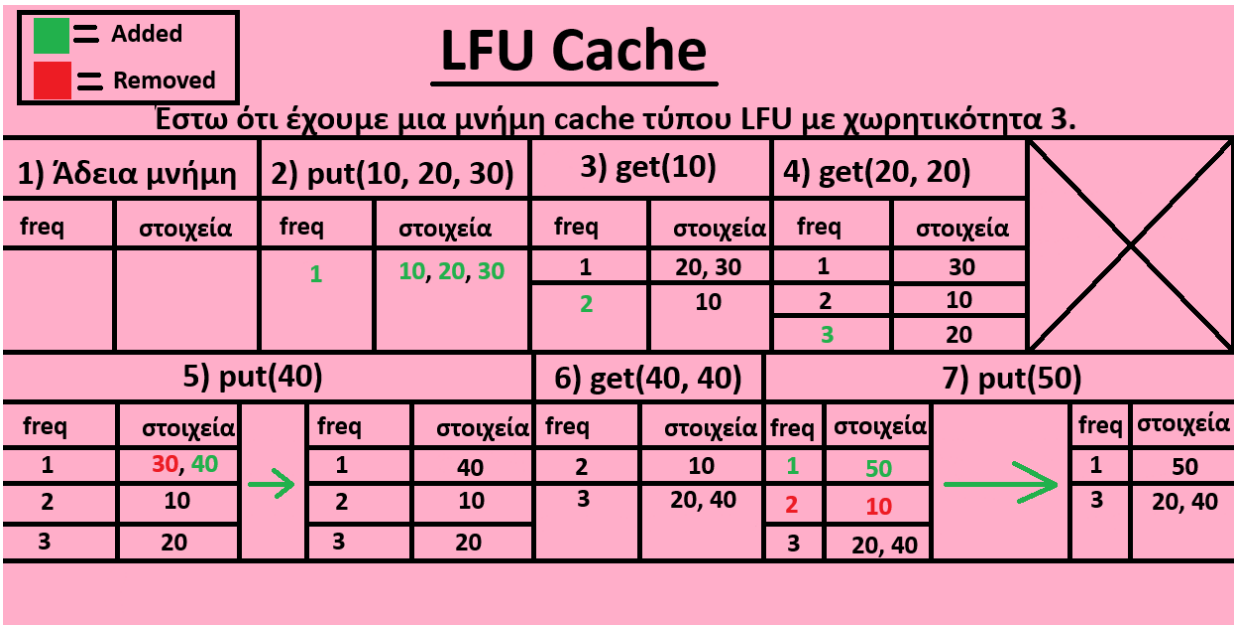
put: Δέχεται ένα κλειδί και μια τιμή και τα προσθέτει στο HashMap με συχνότητα 1. Αν το κλειδί και η τιμή υπάρχουν ήδη στην μνήμη, ανεβάζει κατά 1 την συχνότητα αυτού του κλειδιού αλλιώς αν υπάρχει το κλειδί στην μνήμη αλλά με διαφορετική τιμή, αντικαθιστά την παλιά τιμή με την καινούρια και ανεβάζει κατά 1 την συχνότητα αυτού του κλειδιού.

put (σε γεμάτη μνήμη): Σε περίπτωση που γίνε put σε γεμάτη μνήμη, πριν την εισαγωγή του καινούργιου στοιχείου διαγράφεται από την μνήμη το στοιχείο με την χαμηλότερη συχνότητα (ένα entry του HashMap στο firstKey του TreeMap) για να κάνει χώρο για το καινούργιο στοιχείο και στην συνέχεια ξεκινάει η λειτουργία put.

getHitCount: Επιστρέφει το πλήθος των αναζητήσεων που υπήρχε το στοιχείο που ψάχτηκε στην μνήμη.

getMissCount: Επιστρέφει το πλήθος των αναζητήσεων που δεν υπήρχε το στοιχείο που ψάχτηκε στην μνήμη.

Σχήμα της LFU Cache:



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.

Αυτό το πρόγραμμα υποστηρίζει 3 tests για κάθε μνήμη cache:

1) General test
2) Edge cases test
3) Stress test

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 ελέγχοντας αν η μνήμη βγάζει τα επιθυμητά αποτελέσματα ανάλογα τον τύπο της.

Υλοποίηση test 2 (edge cases test): Ο τρόπος που το edge cases test ελέγχει τις ακριανές τιμές της μνήμης cache είναι ελέγχοντας αν κάθε εισαγωγή καινούργιου στοιχείου αποθηκεύτηκε σωστά και αν διαγράφτηκε το στοιχείο 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 αν τα στοιχεία που έμειναν στην μνήμη ήταν αυτά που έπρεπε να μείνουν ανάλογα με τον τύπο της cache.

3. Λειτουργία Hit/Miss

Η σκοπός της λειτουργίας hit/miss είναι να εμφανίσει πόσες φορές η μνήμη cache περίεχε 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:

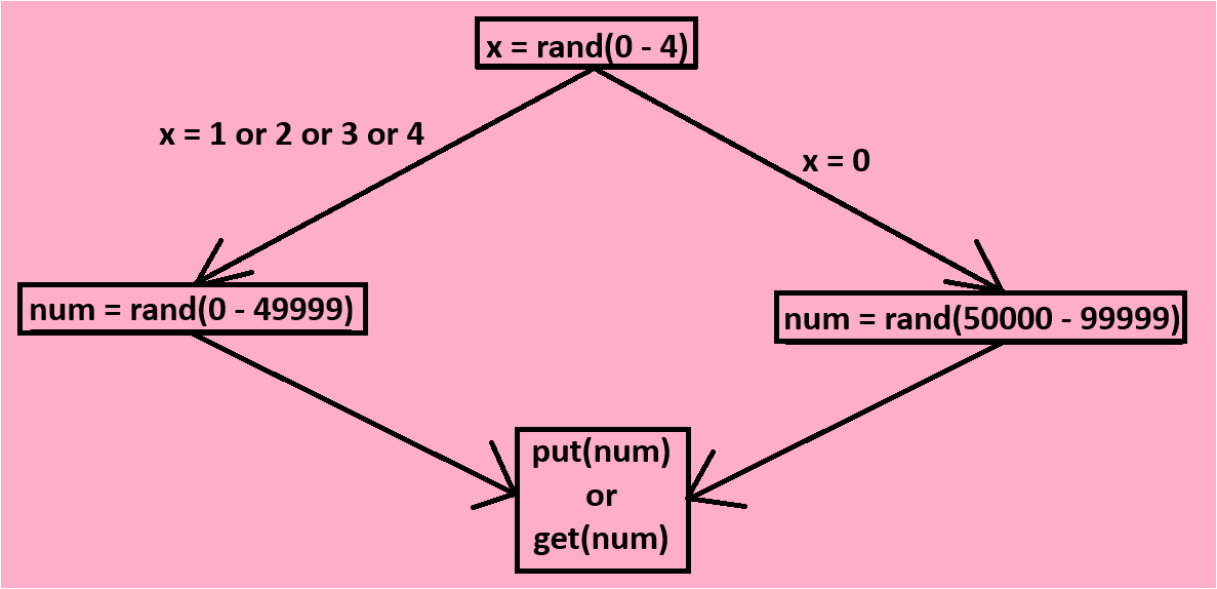
Total operations: 100000
Cache Hits: 52300
Cache Misses: 47700
Hit Rate: 52,30%
Miss Rate: 47,70%

Τρόπος επιλογής στοιχείου:

Θέλουμε να κάθε φορά που κάνουμε put ή get, το στοιχείο που να επιλέγετε να είναι τυχαίο. Παρακάτω εξηγείται ο τρόπος που θα επιλέγονται τα στοιχεία.

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:



Η επιλογή στοιχείου γίνεται με την βοήθεια της Random.

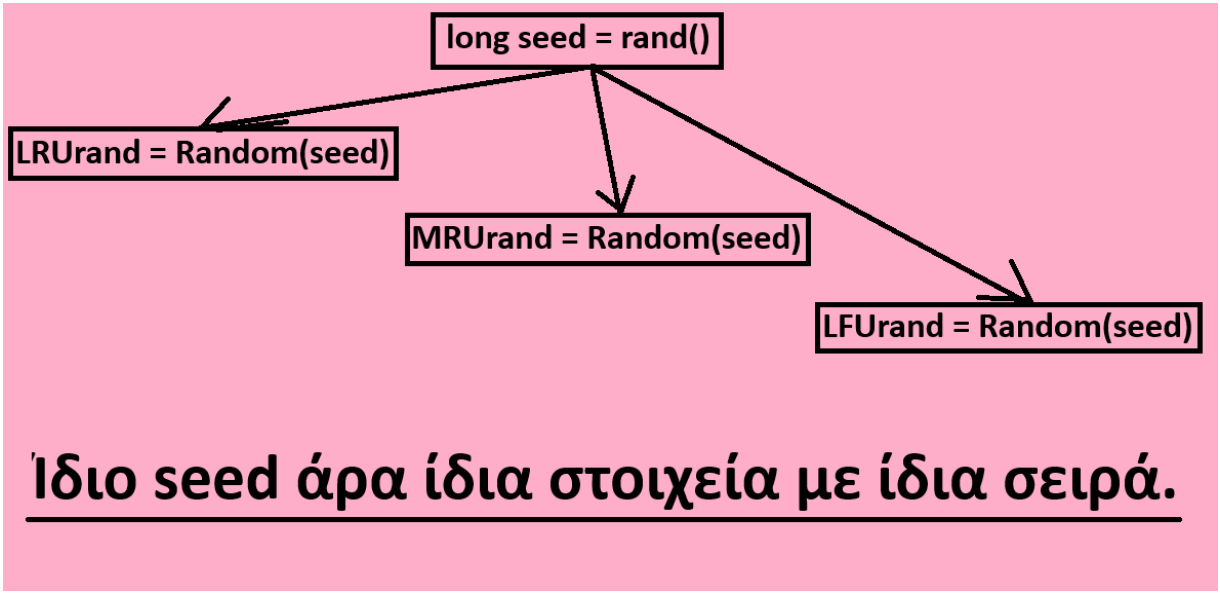
Αντικειμενική σύγκριση Hit/Miss:

Για να μπορεί να υπάρχει αντικειμενική σύγκριση ανάμεσα στα αποτελέσματα της λειτουργίας Hit/Miss για κάθε τύπο μνήμης cache, μια σωστή προσέγγιση θα ήταν για κάθε εκτέλεση του προγράμματος, κάθε μνήμη να δέχεται ακριβώς τα ίδια στοιχεία με ακριβώς την ίδια σειρά ώστε τα αποτελέσματα που θα πάρουμε να βασίζονται σε ίδιες συνθήκες. Πώς όμως θα δίνουμε τα ίδια στοιχεία σε κάθε μνήμη?

Επαναχρησιμοποίηση στοιχείων:

Θέλουμε να κάνουμε επαναχρησιμοποίηση στοιχείων έτσι ώστε κάθε τύπος μνήμης cache να δέχεται τα ίδια στοιχεία. Θα μπορούσαμε να αποθηκεύσουμε κάθε στοιχείο που μας παράγει η Random (σε πίνακα ή λίστα) για να δίνουμε τα ίδια στοιχεία σε κάθε μνήμη, αλλά αυτός ο τρόπος θα μας κόστιζε πολύ μνήμη οπότε δεν τον διαλέγουμε. Άλλος τρόπος είναι για κάθε εκτέλεση του προγράμματος, κάθε φορά που θέλουμε η Random να μας παράγει τυχαία στοιχεία για μια λειτουργία hit/miss, να αρχικοποιείται με το ίδιο seed έτσι ώστε να μας παράγει για κάθε λειτουργία hit/miss τα ίδια στοιχεία με την ίδια σειρά. Επίσης, το seed που αναφέραμε παραπάνω, θα πρέπει για κάθε εκτέλεση του προγράμματος να παράγεται τυχαία από μία άλλη Random έτσι ώστε να παράγουμε διαφορετικά στοιχεία σε κάθε εκτέλεση.

Σχήμα για seed:



Μετά από αρκετές εκτελέσεις παρατηρήθηκε ότι για το συγκεκριμένο πρόγραμμα με την συγκεκριμένη Main το καλύτερο Cache Replacement Policy είναι η LRU, δεύτερο καλύτερο το LFU και χειρότερο το MRU. Αυτή η κατάταξη μπορεί να αλλάξει σε διαφορετικές Main.

Όλες οι εικόνες δημιουργήθηκαν με την βοήθεια του Microsoft Paint.

it2023054