ΕΡΓΑΣΙΑ 4
Μαρία Σχοινάκη: 3210191
Χρήστος Σταμούλος: 3210188

ΑΡΧΕΙΑ

Node.java, Hashing.java, Tcache.java

Στόχος της 4ης εργασίας είναι η δημιουργία μιας γενικού σκοπού υλοποίησης για μια **LRU Cache**. Δεδομένης της ανάγκης προσέγγισης της **βέλτιστης** υλοποίησης σε **χώρο** και **χρόνο** αρχικά σκεφτήκαμε τον συνδυασμό κατακερματισμού και πίνακα. Όμως μειώναμε πολύ την αποδοτικότητα σε χώρο έτσι. Οπότε καταλήξαμε έπειτα από συγκριτική μελέτη των ανάλογων εδαφίων στις διαφάνειες και στο βιβλίο του μαθήματος, στην χρήση του **κατακερματισμού σε συνδυασμό με λίστα διπλής σύνδεσης**.

Αρχίσαμε παράγοντας την κλάση **Node<K, V>**, η οποία αναπαριστά αντικείμενα τύπου κόμβος με τα **K, V generics**. Ένα αντικείμενο τύπου **Node** χαρακτηρίζεται από τα πεδία **previous(προηγούμενος), next(επόμενος), key(κλειδί)** και **value(τιμή)**. Ορίσαμε επίσης τους ανάλογους **setters** και **getters** για να μπορούμε να παίρνουμε τα **private** πεδία της χωρίς να μπορούμε να τα αλλάξουμε, εκτός των **previous, next,** που αλλάζουν στην αντίστοιχη **set**.

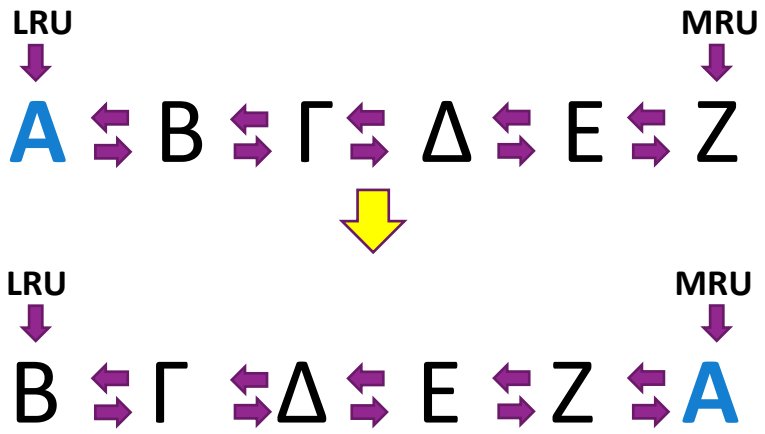
Έπειτα ορίσαμε την κλάση **Hashing<K, V>**. Η κλάση αυτή στην ουσία κατασκευάζει έναν **πίνακα κατακερματισμού**. Ο πίνακας αυτός έχει μέγεθος 5 και έχει υλοποιηθεί έτσι, ώστε να περιέχει ως στοιχείο σε κάθε θέση το πρώτο στοιχείο εισαγωγής στην θέση σύμφωνα με την τιμή της **hash(K key)** συνάρτησης. **Σημαντική σημείωση εδώ είναι ότι τα Nodes που χρησιμοποιεί η κλάση Tcache, είναι διαφορετικά από αυτά της Hashing. Η Hashing, χρησιμοποιεί κόμβους που έχουν σαν V value πεδίο, τους κόμβους της κλάσης Tcache.** Η μέθοδος **hash(K key)**, στην ουσία δέχεται ένα όρισμα και δοσμένης μια συγκεκριμένης συνάρτησης, παράγει έναν ακέραιο από 0 έως 4 που αντιστοιχεί στην θέση που πρέπει να μπει το node με το key στον πίνακα κατακερματισμού. Έπειτα υλοποιήσαμε την μέθοδο **insert(K key, V value)**. (*Το **value** είναι αντικείμενα τύπου **Node** σε αυτήν την κλάση*). Ελέγχουμε προαιρετικά(*Αν και έχει ελεγχθεί στην **TestCacheSpeed***), αν το κλειδί είναι **null** ώστε να μην την υλοποιεί. Έπειτα δημιουργούμε ένα καινούριο **node** με κλειδί το κλειδί της **Tcache** και **value**, το **node** της **cache**. Αυτό το **node** θα αναπαριστά την σχέση με το προηγούμενο και επόμενο στοιχείο που μπαίνει στον πίνακα με ίδιο **hash key**. Καλούμε την **hash**, παίρνουμε το κλειδί που βγάζει σαν αποτέλεσμα(***x***) και ξεκινάμε την διαδικασία. Αν η θέση του πίνακα είναι άδεια, τότε βάζουμε το **node** ως στοιχείο. Αν πάλι δεν είναι, τρέχουμε με μία επανάληψη όλη την λίστα που ξεκινά από το στοιχείο του **HashTable** στην θέση του πίνακα(***HashTable[x]***), φτάνουμε στο τελευταίο(*πιο πρόσφατο στον αριθμοδείκτη x*) και θέτουμε ως **next** με την **setNext()**, αυτού του κόμβου το **node** που θέλουμε να εισάγουμε στον πίνακα κατακερματισμου. Έπειτα υλοποιήσαμε την μέθοδο **remove(K key)**. Η μέθοδος αυτή στην ουσία αφαιρεί το στοιχείο με **key** από τον πίνακα κατακερματισμού (*Στην ουσία απλά το αφαιρεί από την λίστα της θέσης **x***). Αρχικά, με την βοήθεια της **hash(key)**, βρίσκουμε τον κωδικό που αντιστοιχεί στο **key** για τον πίνακα κατακερματισμού. Ελέγχουμε αν το κελί με την θέση του κωδικού είναι άδειο(*Αν είναι επιστρέφουμε **false***) και έπειτα ξεκινάμε την διαδικασία για να βρούμε στην λίστα που αρχίζει από το κελί με τον κωδικό στον πίνακα κατακερματισμού, τον κόμβο με **key** το **key** εισόδου. Αν δεν βρεθεί, η διαδικασία διαγραφής δεν μπορεί να γίνει οπότε επιστρέφουμε **false**. Για να βρεθεί, τρέχουμε μια **while** η οποία με μια **if** ελέγχει αν ο τρέχον κόμβος είναι αυτός που ψάχνουμε. Αν δεν είναι τότε προχωράμε με την **next()**, στον επόμενο. Επίσης σε μία μεταβλητή **prev**, κρατάμε τον προηγούμενο κόμβο από τον τρέχον, έτσι ώστε όταν βρούμε τον κόμβο που πρέπει να διαγραφεί, ο **prev** κόμβος θα πρέπει τώρα να δείχνει στον **next()** του τρέχοντος κόμβου. Στην περίπτωση που ο κόμβος που θέλουμε να διαγράψουμε, είναι

η αρχή της λίστας, δηλαδή το στοιχείο στο κελί του πίνακα κατακερματισμού με θέση τον κωδικό που βρήκαμε, τότε η θέση του πίνακα γεμίζει με τον **next()** του τρέχοντος κόμβου. Έπειτα επιστρέφουμε **true**.Τέλος, υλοποιήσαμε την μέθοδο **get(K key)**, η οποία επιστρέφει έναν κόμβο, που αντιστοιχεί στον κόμβο της κλάσης **Tcache**, που είναι ο τύπος **V(generics)** στην κλάση **Hashing**. Αρχικά, με την βοήθεια της **hash(key)**, βρίσκουμε τον κωδικό που αντιστοιχεί στο **key** για τον πίνακα κατακερματισμού. Ελέγχουμε αν το κελί με την θέση του κωδικού είναι άδειο(*Αν είναι επιστρέφουμε **null***) και έπειτα ξεκινάμε την διαδικασία για να βρούμε στην λίστα που αρχίζει από το κελί με τον κωδικό στον πίνακα κατακερματισμού, τον κόμβο με **key** το **key** εισόδου. Αν δεν βρεθεί, η διαδικασία εύρεσης δεν μπορεί να γίνει οπότε επιστρέφουμε **null**. Για να βρεθεί, τρέχουμε μια **while** η οποία με μια **if** ελέγχει αν ο τρέχον κόμβος είναι αυτός που ψάχνουμε. Αν δεν είναι τότε προχωράμε με την **next()**, στον επόμενο. Αν και όταν τελικά βρεθεί ο κόμβος που έχει κλειδί ίσο με το κλειδί εισόδου, τότε επιστρέφουμε το **value** του, που είναι ο κόμβος της **Tcache**.

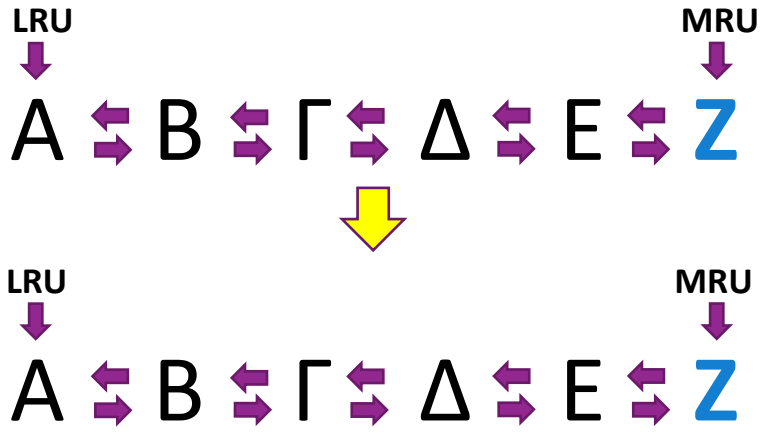
Τέλος, ορίσαμε την κλάση **Tcache<K, V>**. Αρχικά ορίζουμε ένα αντικείμενο τύπου **Hashing<K, Node<K, V>>** για να παριστάνει την **cache,** μια μεταβλητή **LRU** τύπου **Node<K, V>**, που παριστάνει τον λιγότερο πρόσφατο κόμβο που χρησιμοποιήθηκε, μετά μια μεταβλητή **MRU** τύπου **Node<K, V>**, που παριστάνει τον πιο πρόσφατο κόμβο που χρησιμοποιήθηκε και τέλος **4** μεταβλητές που μετράνε πλήθη. Την **cachesize**, που είναι το πλήθος που χωράει η **cache** μας, την **insidecache**, που είναι το τρέχον πλήθος στοιχείων που έχει η **cache**, την **hits**, που μετράει το πλήθος των **hits** της **cache** και τέλος την **miss**, που μετράει το πλήθος των μη **hits** της cache.(***hit=request a key that exists in cache, miss=request a key that doesn't exist in cache***). Όλες αυτές οι μεταβλητές αρχικοποιούνται στον κατασκευαστή. Έπειτα ακολουθεί η μέθοδος **lookup(K key)**. Καλούμε την μέθοδο **get(K key)** της **Hashing** έτσι ώστε να δούμε αν το **node** με κλειδί **key**, υπάρχει στην **cache**. Αν δεν υπάρχει(*δηλαδή η **get(K key)** επιστρέφει **null***), τότε έχουμε **miss**, οπότε αυξάνουμε την μεταβλητή **miss** κατά **1** και επιστρέφουμε **null**. Σε κάθε άλλη περίπτωση έχουμε **hit**, οπότε αυξάνουμε την μεταβλητή **hits** κατά **1**. Έπειτα ελέγχουμε σε τι περίπτωση βρισκόμαστε. Αν ο κόμβος με κλειδί **key** είναι ο ήδη **MRU**, αν είναι ο **LRU** ή αν είναι οποιοσδήποτε άλλος. Αν είναι ο **MRU**, δεν χρειάζεται να αλλάξουμε κάτι γιατί **MRU** είναι ο ίδιος κόμβος που ήταν και πριν το **hit**, οπότε επιστρέφουμε το **value** του. Αν είναι ο **LRU**, τότε θέτουμε τον προηγούμενο κόμβο του επόμενου του τρέχοντος κόμβου σαν **null** και ως **LRU** πλέον τον επόμενο κόμβο του τρέχοντος. Επίσης, θέτουμε σαν προηγούμενο του τρέχοντος κόμβου τον **MRU**, σαν επόμενο κόμβο του **MRU** τον τρέχον κόμβο, έπειτα θέτουμε ως **MRU** τον τρέχον κόμβο και τέλος ως επόμενο του **MRU**(*του τρέχοντος κόμβου δηλαδή*), **null**. Αν ο κόμβος από την άλλη δεν είναι ούτε **MRU** ούτε **LRU,** τότε κάνουμε τα ίδια που κάναμε στον **LRU,** απλά, επειδή τώρα ο τρέχον κόμβος έχει και προηγούμενο που δεν είναι null, τώρα θέτουμε και ως επόμενο του προηγούμενου του τρέχοντος κόμβου, τον επόμενο κόμβο του τρέχοντος κόμβου **και** ως προηγούμενο κόμβου του επόμενου κόμβου του τρέχοντος κόμβου, τον προηγούμενο κόμβο του τρέχοντος. Εν ολίγης στην ουσία οι κόμβοι έχουν προηγούμενο και επόμενο που υποδηλώνει μια χρονική σύνδεση μεταξύ τους. Δηλαδή ποιος χρησιμοποιήθηκε πιο πριν ή πιο μετά από τον άλλον. Οπότε σε κάθε **hit** χρειάζεται και η κατάλληλη αλλαγή ώστε να ισχύει αυτή η σύνδεση. Έπειτα επιστρέφεται το **value**.

ΕΡΓΑΣΙΑ 4
Μαρία Σχοινάκη: 3210191
Χρήστος Σταμούλος: 3210188

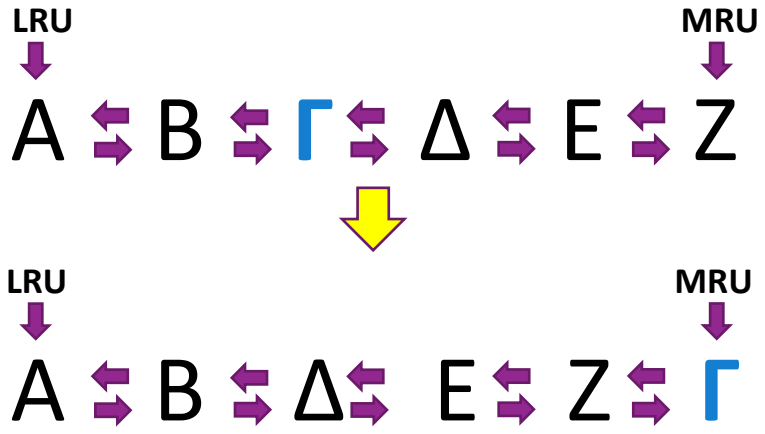
HIT ΣΕ LRU



HIT ΣΕ MRU



HIT ΣΕ ΜΕΣΑΙΟ ΚΟΜΒΟ



ΕΡΓΑΣΙΑ 4
Μαρία Σχοινάκη: 3210191
Χρήστος Σταμούλος: 3210188

Έπειτα υλοποιήσαμε την μέθοδο **store(K key, V value)**, η οποία αποθηκεύει στην **cache** έναν κόμβο με κλειδί και το αντίστοιχο **value** του. Χρησιμοποιούμε την **get(K key)** για να ελέγξουμε αν ο κόμβος που πάμε να προσθέσουμε στην **cache**, υπάρχει ήδη. Αν υπάρχει τερματίζουμε. Αν δεν υπάρχει, ορίζουμε έναν νέο κόμβο ο οποίος θα είναι αυτός που θα προσθέσουμε στην **cache**, με **previous node** τον **MRU**, καθώς στην ουσία αυτός είναι πλέον ο **MRU**. Κάνουμε τον επόμενο κόμβο του **MRU**, τον κόμβο προς αποθήκευση, καλούμε την μέθοδο **insert(key, value)**, όπου σαν **value** στέλνουμε τον κόμβο προς αποθήκευση και τέλος θέτουμε σαν **MRU** τον τρέχον κόμβο. Έπειτα ελέγχουμε αν η **cache** γέμισε με στοιχεία, έτσι ώστε αν είναι να αφαιρούμε το **LRU** στοιχείο, το **LRU** πλέον είναι το επόμενο του προηγούμενου **LRU** και ως προηγούμενο του **LRU** το **null**. Αν η **cache** δεν είναι γεμάτη, αυξάνουμε την μεταβλητή **insidecache** κατά **1** και ελέγχουμε αν το στοιχείο προς εισαγωγή είναι το **1ο** στοιχείο της **cache** και αν είναι θέτουμε αυτό το στοιχείο σαν **LRU**.

Τέλος υλοποιήσαμε **4 getters**. Την **getHits()**, που επιστρέφει τα τρέχοντα **hits**, την **getMisses()**, που επιστρέφει τα τρέχοντα **misses**, την **getNumberOfLookups()**, που επιστρέφει το άθροισμα των **misses** και **hits**, δηλαδή τον συνολικό αριθμό από **lookups** και τέλος την **getHitRatio()**, η οποία επιστρέφει τον συνολικό αριθμό των τρεχόντων **hits** ως προς τον συνολικό αριθμό των **lookups**(*αναλογία των **hits** ως προς τα **lookups***).

Όπως φαίνεται στους παρακάτω πίνακες το πρόγραμμα τρέχει σε **O(n)**, καθώς όσο διπλασιάζουμε τα δεδομένα, διπλασιάζεται περίπου ο συνολικός χρόνος που χρειάζεται για να τρέξει το πρόγραμμα.

Όλες οι μέθοδοι στην κλάση **Node<K, V>**, τρέχουν σε χρόνο **O(1)**, γιατί αποτελούνται από **1** εντολή, είτε ανάθεσης, είτε επιστροφής. Στην κλάση **Hashing<K, V>**, η μέθοδος **hash(K key)**, τρέχει σε χρόνο **O(1)**, καθώς χρησιμοποιεί μόνο μία εντολή ανάθεσης. Η μέθοδος **insert(K key, V value)**, τρέχει σε χρόνο **O(n)**, καθώς χρησιμοποιεί ένα **loop while**, που στην χειρότερη περίπτωση θα περάσει όσα στοιχεία έχουν εισαχθεί(***n***), έχουν ίδιο κωδικό **hashing**, ανήκουν στην ίδια λίστα της θέσης του πίνακα κατακερματισμού με κωδικό **hashing.** Η μέθοδος **remove(K key)**, τρέχει και αυτή σε **O(n)**, αφού, χρησιμοποιεί ένα **loop while**, που στην χειρότερη περίπτωση θα περάσει όσα στοιχεία έχουν εισαχθεί(***n***), έχουν ίδιο κωδικό **hashing**, ανήκουν στην ίδια λίστα της θέσης του πίνακα κατακερματισμού με κωδικό **hashing** και μάλιστα το στοιχείο προς διαγραφή είναι το τελευταίο στοιχείο της λίστας. Τέλος, μέθοδος **get(K key)**, χρησιμοποιεί παρόμοια λογική και **loop** με την **remove(K key)**, άρα τρέχει στον ίδιο χρόνο. Στην κλάση **Tcache<K, V>**, η μέθοδος **lookup(K key)**, τρέχει σε **O(n)**, καθώς χρησιμοποιεί την μέθοδο **get(K key)** της **Hashing<K, V>**, η οποία τρέχει σε **O(n)**, άρα "φορτώνεται" την πολυπλοκότητα της, που είναι και η μεγαλύτερη που θα συναντήσουμε στον αλγόριθμο άρα τρέχει σε **O(n)**. Η μέθοδος **store(K key, V value)**, τρέχει επίσης σε **O(n)**, καθώς χρησιμοποιεί την **get(K key)** της **Tcache**. Βέβαια χρησιμοποιεί και άλλες μεθόδους που φορτώνεται την πολυπλοκότητά τους, όπως την **insert(K key)** και την **remove(K key)**, αλλά είναι και αυτές **O(n)**, αφού όπως ξέρουμε **O(n) + O(n) + O(n) = O(n)**. Τέλος, οι υπόλοιπες **4** μέθοδοι, **getHitRatio(), getHits(), getMisses(), getNumberOfLookups()**, τρέχουν όλες σε **O(1)**, καθώς αποτελούνται από **1** μόνο εντολή, επιστροφής.

ΕΡΓΑΣΙΑ 4

Μαρία Σχοινάκη: 3210191
Χρήστος Σταμούλος: 3210188