Δομές Δεδομένων – Εργασία 4

Βασίλης Παπαδήμας (3220150) και Μάριο Μάτσας (3220120)

Εισαγωγή

Η δομή που χρησιμοποιούμε είναι ένας συνδυασμός Doubly Linked List και HashMap σε μια δομή, που υλοποιείται με την χρήση δεικτών head, tail, left, right και ενός πίνακα. Παράλληλα χρησιμοποιούμε και μια ιδιωτική κλάση Record<K, V>, σαν struct, 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. Για την δομή αυτή εμπνευστήκαμε από το άρθρο https://medium.com/@udaysagar.2177/fastest-lru-cache-in-java-c22262de42ad.

public <thisK, thisV> Falcon(int N)

Οι τύποι αναφέρονται ως thisK και thisV ώστε να αποφεύγεται η επικάλυψη. Ο constructor μας υπολογίζει το totalCapacity του hashtable με βάση scaling (totalCapacity = (int) Math.ceil(N / 0.75)) και δημιουργεί Record objects για όλες τις θέσεις του hashtable, τα οποία δεν καταστρέφονται κατα το τρέξιμο του προγράμματος (συνθήκη null για ένα κελί είναι data[i].key == null και όχι data[i] == null. Αυτό θα μπορούσε να δημιουργήσει πρόβλημα άμα κάποιος ζητήσει να κάνει store ή lookup με null key, αλλά νομίζουμε ότι αυτή η περίπτωση δεν χρειάζεται να καλυφθεί από εμας αφού δεν έχει νοήμα). Αυτό κάνει τον κωδικά μας πιο γρήγορο, επειδή δεν δημιουργούμε συνέχεια νέα αντικείμενα. Για να δημιουργήσουμε generic array με τους τύπους thisK, thisV *at runtime*, χρησιμοποιούμε reflection. Επίσης ενημερώνει την cacheSize σε N, που είναι το πλήθος των στοιχείων που μπορούν να αποθηκευτούν στην cache. Το totalCapacity μπορεί να είναι μεγαλύτερο, αλλά λόγω της υλοποίησης μας το πλήθος των non-null στοιχείων στο hashtable δεν ξεπερνάει ποτέ το N (απλά οι επιπρόσθετες θέσεις μειώνουν την απόσταση του linear probing λόγω spatial locality).

Time complexity: O(N)

Hashing

Για hashing ενός Object, χρησιμοποιούμε το Object.hashcode() % totalCapacity. Στις πιο πρόσφατες εκδόσεις της Java, αυτή η μέθοδος είναι κατανεμημένη αρκετά ομοιόμορφα, και δεν χρειάζεται να εφαρμόσουμε κάποιο bit-mixing[1]. Δοκιμάσαμε επίσης να χρησιμοποιήσουμε την μέθοδο

```
long murmur64(long h) { h ^= h >>> 33; h *= 0xff51afd7ed558ccdL; h ^= h >>> 33;
  h *= 0xc4ceb9fe1a85ec53L; h ^= h >>> 33; return h; }
```

ωστόσο στην πράξη δεν χρειάζεται. Εάν κάποια κλάση δεν έχει υλοποιήσει καλά την μέθοδο hashcode, επιστρέφοντας για παράδειγμα την διεύθυνση του Object (π.χ. στο String υπολογίζεται ως εξής: `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`), τότε η κατανομή είναι skewed λόγω memory alignment και αυτή η μέθοδος χρειάζεται όντως. Ωστόσο, πιστεύουμε ότι αυτό είναι λάθος του

[1] https://sigpwned.com/2018/08/10/string-hashcode-is-plenty-unique/

υλοποιητή της κλάσης, καθώς σύμφωνα με το contract του Object[2], “This method is supported for the benefit of hash tables such as those provided by HashMap”. Σημειώνουμε ότι κανουμε το hashing inline ώστε να μην δημιουργούνται call stack frames κάθε φορά που χρειαζόμαστε hash. Επίσης, έχουμε αφήσει commented-out την μέθοδο hash() βασισμένη στην murmur64, και μπορεί να χρησιμοποιηθεί αντί του απλού hashcode αλλάζοντας τις γραμμές 47, 74, 154, 163 και 210.

Γιατί Linear Probing?

Είναι γεγονός ότι η υλοποίηση ενός HashMap με την χρήση γραμμικής διερεύνησης είναι πολύ πιο εύκολη σε σχέση με την χωριστή αλυσίδωση, ωστόσο αυτό από μόνο του δεν αποτελεί σοβαρό λόγο για να την προτιμήσουμε. Αν όμως το καλοσκεφτούμε η γραμμική διερεύνηση βγάζει πολύ περισσότερο νόημα για την υλοποίηση μιας cache, αφού ο χώρος μνήμης της είναι ούτως ή άλλως περιορισμένος και **συνεχόμενος** και άρα δεν θα ήταν λογικό να χρησιμοποιήσουμε χωριστή αλυσίδωση, καθώς η μέθοδος αυτή χρησιμοποιεί pointers που μπορούν να δείχνουν οπουδήποτε στην μνήμη, χειροτερεύοντας έτσι το spatial locality της cache μας, ενώ η γραμμική διερεύνηση λειτουργεί τέλεια μιας και έχουμε συνεχόμενη μνήμη. (Γενικά σε τέτοιες περιπτώσεις η χρήση γραμμικής διερεύνησης υπερτερεί της χωριστής αλυσίδωσης **δεδομένου ότι δεν παρουσιάζεται clustering**).

public V lookUp(K key)

Αφού χρησιμοποιούμε το linear probing, αν στο κελί data[hash] δεν υπάρχει το στοιχείο, τότε σίγουρα δεν είναι στην cache, αλλά αν στο κελί υπάρχει κάποιο στοιχείο με άλλο key (συνέβη collision) τότε μπορεί να έχει αποθηκευτεί κάπου αλλού στην cache το key που αναζητούμε, άρα κάνουμε γραμμική αναζήτηση. Λόγω spatial locality και των έξτρα θέσεων που έχουμε στο hashtable, αν υπάρχει όντως το key στην cache συνήθως η γραμμική αναζήτηση δεν θα διαρκέσει πολύ. Στην χειρότερη περίπτωση φυσικά (είτε υπάρχει είτε όχι) θα διατρέξει όλη την cache. Εάν βρούμε το key, κάνουμε remove και add ώστε να ενημερώσουμε το priority (γίνεται το tail της λίστας).

Time complexity: O(1) average case, O(N) worst case[3]

public void store(K key, V value)

Χρησιμοποιούμε δύο loops, αλλά τρέχει πάντα μόνο ένα. Αν η cache είναι γεμάτη αφαιρούμε το least recently used στοιχείο, κάνουμε shift keys και τρέχει το πρώτο loop, όπου διασχίζουμε την cache αρχίζοντας από το κλειδί που δείχνει το hash, μέχρι να βρούμε το πρώτο κενό κελί για αποθήκευση (που είναι εγγυημένο ότι θα βρεθεί), οπότε το εισάγουμε και επιστρέφουμε. Αν η cache δεν έχει γεμίσει, στο δεύτερο loop διασχίζουμε γραμμικά το hashtable, αρχίζοντας από το κελί που δείχνει το hash, μέχρι να γίνει return είτε εάν το key ήταν ήδη στο data[hash(key)], στην οποία περίπτωση ενημερώνουμε το value και γίνεται το tail της λίστας, ή αν data[hash].key == null δηλαδή το κελί που δείχνει το hash είναι ελεύθερο, άρα απλά εισάγουμε το key. Και τα δύο loops έχουν συνθήκη while (true), αλλά στην πραγματικότητα θα γίνουν το πολύ totalCapacity = O(N) επαναλήψεις, επειδή εκτελούμε το κάθε loop κάτω από συγκεκριμένες συνθήκες που εγγυούν ότι θα τερματίσει πριν φτάσει ξανά στο σημείο που ξεκίνησε η διάσχιση.

---

[2] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html#hashCode()
[3] https://web.archive.org/web/20160303225949/http://algo.inria.fr/AofA/Research/11-97.html

Time complexity: O(1) average case, O(N) worst case[3]

void shift(int pos)

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 add(int pos)

Μέσω 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 remove(int pos)

Μέσω 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