Authors: Georgios Patrikis (3160141), Georgios Fotopoulos (3130220)

Date: 24/02/2018

Subject: Data Structures – Project 4

E-mail: geopatrikis12@gmail.com, giorgos.fotopoulos7@gmail.com

ΓΕΝΙΚΗ ΙΔΕΑ

Ύστερα από αρκετή αναζήτηση στις σημειώσεις του μαθήματος και το διαδίκτυο, αποφασίσαμε να χρησιμοποιήσουμε για την υλοποίηση της Cache δύο δομές δεδομένων, την Ουρά Προτεραιότητας (Max Heap) και τον Κατακερματισμό με Γραμμική Διερεύνηση (Linear Hash Table). Από τη στιγμή που ο ελάχιστος χρόνος είναι το ζητούμενο, θυσιάζοντας μερικούς πόρους μνήμης η γραμμική διερεύνηση μας δίνει αναζήτηση σε O(1) πολυπλοκότητα. Σύμφωνα με τις σημειώσεις του μαθήματος, η γραμμική διερεύνηση είναι πιο γρήγορη από τον διπλό κατακερματισμό. Η χωριστή αλυσίδωση, είναι πιο memory-efficient, αλλά όχι πιο γρήγορη από τους παραπάνω δύο τρόπους. Βέβαια μπορεί η αναζήτηση να γίνεται σε O(1), όμως θα πρέπει να ενημερώνεται η προτεραιότητα των στοιχείων. Αυτό μπορεί να επιτευχθεί μέσω μιας Max Heap, στην οποία όλες οι διεργασίες γίνονται με πολυπλοκότητα O(logN). Σε περίπτωση εισαγωγής, ο χρόνος εισαγωγής στο Hash Table είναι O(1) και εφόσον το Max Heap δεν είναι γεμάτο και πάλι O(1). Αν αυτό γεμίσει, τότε κάθε εισαγωγή απαιτεί O(logN) χρόνο, καθώς θα πρέπει να γίνει εξαγωγή του παλαιότερου στοιχείου και αντικατάσταση της κορυφής με το επόμενο χρονικά στοιχείο. Θεωρήσαμε πως αυτές οι δομές είναι σε συνδυασμό οι πιο αποδοτικές, καθώς σε καμία λειτουργία δεν ξεπερνιέται η πολυπλοκότητα O(logN). Δε μπορούμε να χρησιμοποιήσουμε αποκλειστικά μια δομή (πχ. Hash), καθώς οι αναζητήσεις για το παλαιότερο στοιχείο θα είχαν πολύ μεγάλο κόστος. Οποιαδήποτε άλλη ουρά προτεραιότητας εκτός της heap θα λειτουργούσε με O(N) σε τουλάχιστον μια διεργασία.

ΥΠΟΛΟΓΙΣΤΙΚΟ ΚΟΣΤΟΣ

1. *Εισαγωγή:* Η εισαγωγή στην Cache γίνεται με πολυπλοκότητα O(logN) . Το στοιχείο που εισάγεται πρώτα μπαίνει στην Max Heap. Η εισαγωγή στην MH γίνεται με κόστος O(1) όταν ο πίνακας δεν είναι γεμάτος, όμως όταν γεμίσει θα πρέπει να αφαιρεθεί το στοιχείο που βρίσκεται στην κορυφή. Αυτό θα πάρει O(logN) χρόνο και στην συνέχεια στην θέση που γίνεται delete το στοιχείο θα γίνει replace το νέο στοιχείο. Στην εισαγωγή στο Hash table όλα τα αντικείμενα θέλουν χρόνο μερικών επαναλήψεων ανάλογα με τον αριθμό των συγκρούσεων. Τελικά, η γενικότερη συμπεριφορά μάλλον καθορίζεται από την εισαγωγή στην Max Heap όταν ο πίνακας είναι γεμάτος και έτσι η πολυπλοκότητα υπολογίζεται σε O(logN). Σημείωση: Το delete με το Rehash δεν φαίνεται να έχουν σημαντικό κόστος σε σχέση με την εισαγωγή.

2. Αναζήτηση: Η αναζήτηση επιτυγχάνεται σε λίγες μόλις επαναλήψεις μέσω του Hash Table με βάση το key που δίνεται και της hash μεθόδου γίνεται αναζήτηση στον πίνακα και με βήμα +1 ελέγχουμε αν

υπάρχει το στοιχείο. Βέβαια η αναζήτηση αποκτά μεγαλύτερο κόστος εξαιτίας του update που πρέπει να κάνει στον χρόνο που χρησιμοποιήθηκε τελευταία φορά το αντικείμενο. Έτσι έχουμε O(logN) χρόνο στην χειρότερη περίπτωση (Αν κάνουμε αναζήτηση στο πιο παλιό στοιχείο της Heap), που προκύπτει από την sink(). Τελικά ο χρόνος της αναζήτησης θεωρείται O(logN) αν λάβουμε υπόψιν και την ενημέρωση των στοιχείων.

ΔΟΜΗ ΠΡΟΓΡΑΜΜΑΤΟΣ

CacheImpl

Η υλοποίηση της διεπαφής για τη δομή μιας μνήμης Cache.

public CacheImpl(int cacheSize): Κατασκευαστής της Cache, παίρνει σαν παράμετρο το cacheSize που είναι το μέγεθός της.

public V lookUp(K key): Μέθοδος που αναζητά και επιστρέφει τα δεδομένα V που σχετίζονται με το κλειδί K που δίνεται σαν παράμετρος.

public void store(K key, **V** value): Μέθοδος που αποθηκεύει τα δεδομένα που σχετίζονται με το δοθέν κλειδί. Αν υπάρχει ήδη το κλειδί στον Πίνακα Κατακερματισμού, τότε διαγράφονται τα παλιά δεδομένα και εισάγονται τα καινούρια.

public double getHitRatio(): Επιστρέφει το ποσοστό των επιτυχών αναζητήσεων της Cache, δηλαδή το αποτέλεσμα του κλάσματος «Επιτυχείς Αναζητήσεις / Συνολικές Αναζητήσεις».

public long getHits(): Επιστρέφει τον απόλυτο αριθμό των επιτυχών αναζητήσεων της Cache.

public long getMisses(): Επιστρέφει τον απόλυτο αριθμό των αποτυχημένων αναζητήσεων της Cache.

public long getNumberOfLookUps(): Επιστρέφει τον συνολικό αριθμό αναζητήσεων της εν λόγω Cache.

CacheSlice<K,V>

Η κλάση αυτή αφορά τα αντικείμενα που μπαίνουν σε μια μνήμη Cache. Παίρνει τα generic type K, V, όπου K – Key και V – value.

public CacheSlice(K key, **V** value): Κατασκευαστής ενός κομματιού Cache, θέτει τις τιμές του κλειδιού και των δεδομένων ίσες με αυτές που δίνονται σαν παράμετροι, καθώς επίσης δίνει τιμή στο χρόνο του αντικειμένου ίσο με μια στατική μεταβλητή τύπου long, ονόματι timestamp που κρατάμε στην κλάση, η οποία μεταβάλλεται για κάθε αντικείμενο που δημιουργούμε ή για κάθε κλήση της updateTime(), καθώς και την default τιμή στον δείκτη.

public void updateTime(): Μέθοδος που μεταβάλλει τη τιμή του χρόνου ενός κομματιού της Cache.

public void setValue(V value): Μέθοδος που θέτει τα δεδομένα ενός κομματιού της Cache ίσο με τη τιμή που δίνεται σαν παράμετρος.

public V getValue(): Μέθοδος που επιστρέφει τα δεδομένα του αντικειμένου.

public void setIndex(int index): Μέθοδος που θέτει το δείκτη ενός κομματιού της Cache ίσο με τη τιμή που δίνεται σαν παράμετρος.

public int getIndex(): Μέθοδος που επιστρέφει το δείκτη του αντικειμένου.

public void setTime(long time): Μέθοδος που θέτει το χρόνο ενός κομματιού της Cache ίσο με τη τιμή που δίνεται σαν παράμετρος.

public long getTime(): Μέθοδος που επιστρέφει το χρόνο του αντικειμένου.

public void setKey(K key): Μέθοδος που θέτει το κλειδί ενός κομματιού της Cache ίσο με τη τιμή που δίνεται σαν παράμετρος.

public K getKey(): Μέθοδος που επιστρέφει το κλειδί του αντικειμένου.

LinearHashTable

Η κλάση αυτή αναπαριστά τη δομή του Πίνακα Κατακερματισμού.

LinearHashTable(int capacity): Ο κατασκευαστής του πίνακα κατακερματισμού. Δέχεται σαν όρισμα έναν integer με όνομα capacity, ο οποίος ορίζει το μέγιστο μέγεθος και πλήθος στοιχείων που μπορεί να δεχθεί. Κατά τη δημιουργία του πίνακα αυτού, δημιουργούμε παράλληλα και μία ουρά προτεραιότητας.

public int size(): Μέθοδος που επιστρέφει τον αριθμό των στοιχείων στον πίνακα.

boolean isEmpty(): Μέθοδος που ελέγχει αν ο πίνακας είναι άδειος από στοιχεία, αν ναι επιστρέφει true, διαφορετικά επιστρέφει false.

public V LookUp(K key): Αυτή η μέθοδος δέχεται ένα κλειδί σαν όρισμα, το οποίο αφού περάσει από τη μέθοδο hash(K) ψάχνει στον πίνακα κατακερματισμού. Αν βρεθεί το κλειδί σε αυτόν, τότε επιστρέφεται η τιμή του, διαφορετικά επιστρέφεται η τιμή null. Αν βρεθεί το κλειδί, πρέπει επίσης να μεταβληθεί ο χρόνος του συγκεκριμένου αντικειμένου με τη βοήθεια της updateTime(), καθώς και να ανανεωθεί η MaxHeap μας με την χρήση της μεθόδου sink(int).

private int hash(K key): Η μέθοδος κατακερματισμού που χρησιμοποιούμε για τα generic types.

public void Store(K key, V value): Αυτή η μέθοδος δημιουργεί ένα αντικείμενο τύπου CacheSlice, χρησιμοποιώντας τις τιμές K, V που δίνονται σαν παράμετροι, και ύστερα καλεί την μέθοδο put(CacheSlice<K,V>, boolean) προκειμένου να γίνει η εισαγωγή του αντικειμένου.

public void put(CacheSlice<K,V> element, boolean flag): Αυτή η μέθοδος εισάγει ένα αντικείμενο τύπου CacheSlice στον πίνακα, το οποίο δίνεται σαν όρισμα. Επίσης έχει ένα όρισμα τύπου boolean, με όνομα flag και είναι τιμή φρουρός, και αλλάζει ανάλογα με τις ανάγκες του προγράμματος. Στην συγκεκριμένη μέθοδο, αν δοθεί η τιμή true, τότε το στοιχείο εισάγεται και στην Max Heap μας και ύστερα τίθενται οι ανάλογες τιμές για τον δείκτη και τον χρόνο του στοιχείου. Αν η ουρά προτεραιότητας είναι γεμάτη προτού γίνει η προσθήκη, τότε αφαιρείται το παλαιότερο στοιχείο της και προστίθεται το καινούριο.

public void delete(K key): Η μέθοδος αυτή δέχεται ένα κλειδί σαν όρισμα, ψάχνει τη θέση του στον πίνακα, διαγράφει την τιμή του και ύστερα αλλάζει την θέση όλων των στοιχείων που το ακολουθούν μέχρι να συναντήσει την τιμή null σε κάποια θέση.

MaxHeap

Η κλάση αυτή αναπαριστά τη δομή της Ουράς Προτεραιότητας.

public MaxHeap(int capacity): Ο κατασκευαστής της ουράς προτεραιότητας. Δέχεται σαν όρισμα έναν integer με όνομα capacity, ο οποίος ορίζει το μέγιστο μέγεθος και πλήθος στοιχείων που μπορεί να δεχθεί η ουρά.

public int getCap(): Η μέθοδος αυτή επιστρέφει τη χωρητικότητα της ουράς.

private boolean isFull(): Μέθοδος που επιστρέφει true αν η ουρά έχει γεμίσει από αντικείμενα και false αν όχι.

private int parent(int pos): Αυτή η μέθοδος επιστρέφει τον γονιό του node που βρίσκεται στη θέση που δόθηκε σαν όρισμα.

private void swap(int fpos, **int** spos): Αυτή η μέθοδος ανταλλάσσει τις θέσεις των δύο node που δίνονται σαν ορίσματα.

public int insert(CacheSlice<K,V> element): Αυτή η μέθοδος εισάγει το αντικείμενο που δίνεται σαν παράμετρος στην ουρά προτεραιότητας. Αν αυτή είναι γεμάτη, τότε το παλαιότερο αντικείμενο, αυτό δηλαδή που βρίσκεται στην πρώτη θέση, είναι αυτό που θα αφαιρεθεί από την ουρά, ώστε το αμέσως παλαιότερο να πάρει τη θέση του. Αυτό γίνεται με τη κλήση της sink(int). Ύστερα από την διαδικασία της κατάδυσης, αφαιρείται αυτό το στοιχείο και το νέο παίρνει τη θέση αυτού. Αν η ουρά δεν ήταν γεμάτη, τότε γίνεται μια απλή εισαγωγή του νέου στοιχείου στην επόμενη διαθέσιμη θέση. Σε κάθε περίπτωση, επιστρέφεται ο δείκτης της θέσης στην οποία το στοιχείο εισάχθηκε.

public void sink(int pos): Αυτή η μέθοδος παίρνει σαν όρισμα τον δείκτη της θέσης στην οποία βρίσκεται το αντικείμενο, το οποίο θέλουμε να καταδυθεί, ώστε να φτάσει στην θέση που πρέπει, προκειμένου να εκτελεστεί η διαδικασία που χρησιμοποιεί αυτή τη μέθοδο, δηλαδή η LookUp(K). Αυτό επιτυγχάνεται με διαδοχικές κλήσεις της swap(int, int), όπου το στοιχείο αυτό κάθε φορά αλλάζει θέση με το μικρότερο από τα δύο παιδιά του, το αριστερό και το δεξί.

private int replace(int pos, **CacheSlice**<K,V> newElement): Αυτή η μέθοδος αποτελεί μια παραλλαγή της sink(int), όπου ουσιαστικά παίρνουμε σαν όρισμα τον δείκτη της θέσης που βρίσκεται το αντικείμενο που θέλουμε να αντικαταστήσουμε με το νέο αντικείμενο που δίνεται σαν όρισμα, εκτελείται πάλι ο αλγόριθμος με τις εναλλαγές θέσεων μεταξύ αυτού του αντικειμένου και των δύο παιδιών του, και μόλις αυτό φτάσει στην κατάλληλη θέση, αλλάζουμε τον δείκτη του και στη θέση αυτή βάζουμε το νέο αντικείμενο, του οποίου τη θέση επιστρέφουμε.

ΠΗΓΕΣ

- https://en.wikipedia.org/wiki/Open_addressing
- https://en.wikipedia.org/wiki/Binary_heap
- https://cs.stackexchange.com/questions/10273/hash-table-collision-probability
- http://tutorials.jenkov.com/java-collections/hashcode-equals.html (hash-Code)
- Διαφάνειες Μαθήματος Linear αντί double (κεφ. 15 slide 32)
- Διαφάνειες Μαθήματος για τις υλοποιήσεις (κεφ. 11 ,15)