Project report - Εργασία 4.

Η δομή δεδομένων που χρησιμοποιήθηκε στην υλοποίηση της cache δεν είχε ξεκάθαρη μορφή.  
Είναι βασισμένη στο συνδυασμό ενός πίνακα κατακερματισμού χωριστής αλυσίδωσης και μιας λίστας διπλής σύνδεσης.

Ο λόγος που η μορφή της δομής δεν είναι ξεκάθαρη είναι γιατί όταν υλοποιήθηκε με modular μορφή, (ξεχωριστή υλοποίηση για τη λίστα διπλής σύνδεσης για τα timestamps, ξεχωριστή υλοποίηση για τον κουβά που υπάρχει σε κάθε κελί του πίνακα κατακερματισμού και ξεχωριστή υλοποίηση για τον πίνακα κατακερματισμού) ο μέσος χρόνος εκτέλεσης της lookUp αυξάνονταν, καθώς και υπήρχαν έξτρα επιβαρύνσεις όπως η αρχικοποίηση κάποιων δομών.  
  
Πηγή για επιλογή δομής :   
<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8bcd0ae2891c22245f9a62f63a1698fe004712a9>

Πίνακας κατακερματισμού.

Αρχικά η ιδέα ήταν να υπάρχει μέσα σε κάθε κελί του πίνακα μία κλασσική λίστα, η οποία σε κάθε κόμβο θα είχε και μία αναφορά σε κόμβο της λίστας διπλής σύνδεσης της cache, έτσι ώστε να μπορούν οι προσθήκες, διαγραφές και ευρέσεις σε χρόνο Ο(1).  
Εν τέλη η τελική υλοποίηση είναι ο κάθε “κουβάς” αντί για ολόκληρη λίστα να είναι ένα HashNode το οποίο είναι συνδεδεμένο με τα υπόλοιπα HashNodes του κουβά (δηλαδή αντί για όλη την λίστα, κάθε κελί του πίνακα είχε απλά το κεφάλι της λίστας). Αυτό μείωσε το μέσο όρο εύρεσης των υπαρχόντων στοιχείων κατά περίπου 15% και γλίτωσε το χρόνο αρχικοποίησης κάθε λίστας σε κάθε κελί του πίνακα.

Λίστα διπλής σύνδεσης.

Η υλοποίηση για να είναι αποτελεσματική έγινε λίγο πιο περίπλοκη εδώ.

Η CacheLRU , περιέχει τον πίνακα κατακερματισμού, όμως αντί για την τιμή που αντιστοιχεί στο κλειδί, η υλοποίηση του πίνακα έχει σαν τιμή V το αντικείμενο

class LinkedValue {

Κ key;

V value;

LinkedValue previous, next;

long timestamp;

}

Με αυτόν τον τρόπο έχουμε στην ουσία μια εμφωλευμένη δομή διπλά συνδεδεμένης λίστας μέσα στην υλοποίηση της Cache, και τα αντικείμενα που αντιστοιχούν στις τιμές των κλειδιών είναι συνδεδεμένα άμεσα με τα επόμενα/προηγούμενα έτσι ώστε να είναι εύκολη η μετατόπιση, απαλοιφή, εισαγωγή τους.   
  
Η υλοποίηση βασίζεται στην ιδέα:   
https://www.interviewcake.com/concept/java/lru-cache

Αναλυτικά οι δοκιμές που έγιναν :

10k requests

| Time Averages | Final | Linear Probing | Modular 1 | Modular 2 | HashMap - BST | HashMap Integrated BST |
| --- | --- | --- | --- | --- | --- | --- |
| Execution | 1750 ms | 1780 ms | 1760 ms | 1760 ms | 1800 ms | 1820 ms |
| Lookup | 280 ns | 1520 ns | 340 ns | 330 ns | 330 ns | 320 ns |
| Store | 760 ns | 2790 ns | 890 ns | 880 ns | 1010 ns | 980 ns |

100k requests

| Time Averages | Final | Linear Probing | Modular 1 | Modular 2 | HashMap - BST | HashMap Integrated BST |
| --- | --- | --- | --- | --- | --- | --- |
| Execution | 43190 ms | 45420 ms | 43760 ms | 44120 ms | 43430 ms | 44560 ms |
| Lookup | 260 ns | 4890 ns | 340 ns | 300 ns | 320 ns | 310 ns |
| Store | 720 ns | 7310 ns | 760 ns | 590 ns | 700 ns | 670 ns |

Πιο συγκεκριμένα, τα αναφερόμενα “κλαδιά” του προγράμματος που εξερευνήθηκαν έχουν ως εξής :

Final : Η τελική μορφή, που παρουσιάστηκε παραπάνω.

Linear Probing : Η Final, απλά με linear probing αντί για separate chaining.

Modular 1,2 : Δύο διαφορετικές μορφές της Final, (ελάχιστες μικροδιαφορές μεταξύ τους για λόγους βελτιστοποίησης). Η βασική διαφοροποίηση είναι ότι οι υλοποίησεις των δύο λιστών (του κουβά για το hash table) και της διπλά συνδεδεμένης λίστας για τα timestamps, είναι διακριτές, σε ξεχωριστά αρχεία αντί να είναι κομμάτι του HashMap και της Cache, αντίστοιχα.

HashMap - BST : αντίστοιχο της Final, η μόνη διαφορά είναι ότι αντί για λίστα με δείκτες σε αντικείμενα, οι κουβάδες του HashMap είναι διακριτά δέντρα δυαδικής αναζήτησης.

Hashmap - integrated BST : Αντίστοιχο της Final, αλλά αντί για λίστα με δείκτες σε κεφαλές λιστών, οι δείκτες δείχναν σε ρίζες δέντρων. Αξίζει να σημειωθεί ότι για διαφορετική αναλογία μεγέθους cache/κουβάδων, η εφαρμογή με ΔΔΑ θα ήταν καλύτερη. Δε δημιουργούνται όμως με τα δεδομένα αρκετά μεγάλες λίστες λίστες (το μέγιστο μήκος ήταν 15, και ο μέσος όρος 10). Ο έξτρα χρόνος εν τέλη, που απαιτούν όλες οι διενέργειες για την αποκατάσταση του δέντρου ξεπερνούν τη διαφορά εύρεσης στη λίστα O(n) έναντι του O(logn).

Βάση των παραπάνω πινάκων, που προέκυψαν μετά από 5 και 2 τρεξίματα του κάθε προγράμματος, αντίστοιχα για 10k και 100k requests, βλέπουμε ότι η μόνη υλοποίηση που έρχεται κοντά στην μορφή που τελικά επιλέχθηκε, είναι αυτή του δδα.

Τελική σημείωση:

Λόγω παλαιότητας θα ήθελα αν γίνεται να μη συμπεριληφθεί στα αποτελέσματα σύγκρισης η εργασία γιατί πιστεύω πως θα ήταν άδικο για τους νεότερους φοιτητές με λιγότερη εμπειρία.