Χρήστος Κωσταδήμας 2020030050 *Εαρινό Εξάμηνο 2021-2022*

Η main μέθοδος υπάρχει στην κλάση MainClass που περιέχεται στο πακέτο org.tuc.test. Το πακέτο org.tuc.counter περιέχει την κλάση MultiCounter χωρίς καμία αλλαγή, ακριβώς όπως αυτή δόθηκε στο 3ο φροντιστήριο του μαθήματος. Εξωτερικές πηγές πληροφόρησης που χρησιμοποιήθηκαν για διευκόλυνση παρατίθενται στο τέλος αυτής της αναφοράς. *Το πρόγραμμα μεταγλωττίζεται με επιτυχία και απαντά σε όλα τα ερωτήματα χωρίς παραγωγή λαθών ή παράλογες χρονικές καθυστερήσεις.*

ΜΕΡΟΣ Α : Επεξεργασία στην Κεντρική Μνήμη

Ερώτημα Α.1 : Συνδεδεμένες Λίστες

Μετρούνται μόνο συγκρίσεις μεταξύ (x,y). Δηλαδή ο έλεγχος (x1,y1)?=(x2,y2) αυξάνει τον μετρητή συγκρίσεων κατά ένα.

Class Node (org.tuc): αναπαριστά τα στοιχεία της συνδεδεμένης λίστας. Οι μεταβλητές που φέρει είναι δύο integers x και y και μια αναφορά Node next η οποία κρατά το επόμενο στοιχείο της συνδεδεμένης λίστας.

Class List (org.tuc.list) implements interface Search (org.tuc): Διαθέτει 3 μεταβλητές: Node head (κεφαλή λίστας), Node tail (τελευταίο στοιχείο λίστας), boolean flag. Η τελευταία χρησιμοποιείται για να δηλώσει αν η αναζήτηση στοιχείου ήταν επιτυχής ή όχι. Με κατάλληλο override οι void add(Node node) και void add(int x, int y) αξιοποιώντας την αναφορά tail προσθέτουν ένα στοιχείο στο τέλος της λίστας χωρίς να τη διασχίσουν. Οι μέθοδοι long search(Node node), long search(int x, int y) διασχίζουν τη λίστα μέχρι να βρεθεί το επιθυμητό στοιχείο και επιστρέφουν τον αριθμό συγκρίσεων που απαιτήθηκαν για την αναζήτηση.

Ερώτημα Α.2 : Μέθοδος Κατακερματισμού (ο τρόπος λύσης βασίστηκε στο μέρος Α.1)

Class Hash (org.tuc.hash) implements interface Search (org.tuc): Διαθέτει 2 μεταβλητές: List[] hashTable και boolean flag. Το array hashTable περιέχει τις συνδεδεμένες λίστες του πίνακα κατακερματισμού και η μεταβλητή flag δηλώνει αν η αναζήτηση ήταν επιτυχής ή όχι. Το μέγεθος του πίνακα είναι M=10. Οι μέθοδοί void add(Node node) και void add(int x, int y) πρώτα καλούν την μέθοδο static int hashFunc (int x, int y) της ίδιας κλάσης για τον υπολογισμό της θέσης pos του πίνακα όπου προορίζεται να προστεθεί το νέο στοιχείο , στη συνέχεια καλούν τη μέθοδο add της κλάσης List για την ολοκλήρωση της προσθήκης. Οι μέθοδοι long search(Node node) και long search(int x, int y) μέσω της hashFunc υπολογίζουν τη θέση του πίνακα που περιέχει την λίστα όπου πρέπει να γίνει η αναζήτηση του node, με την μέθοδο search της κλάσης List ,η οποία θα επιστρέψει τον αριθμό συγκρίσεων που απαιτήθηκαν.

ΜΕΡΟΣ Β : Επεξεργασία στον Δίσκο

Το μέγεθος μίας σελίδας δίσκου είναι 256bytes και ένα ζευγάρι ακεραίων (x,y) καταλαμβάνει 8 bytes επομένως μια σελίδα χωράει ακριβώς 32 ζευγάρια ακεραίων (x,y).

Ερώτημα Β.1 : Συνδεδεμένες λίστες στον δίσκο

Class DiskList (org.tuc.list): Βασικές μεταβλητές της κλάσης είναι οι: buffer byte[] dataPage μεγέθους 256 που χρησιμοποιείται για την αποθήκευση μίας σελίδας δίσκου στην κεντρική μνήμη και το byte[] pair = byte[8] που θα περιέχει ένα ζευγάρι ακεραίων σε μορφή byte. Ακολουθούν οι μέθοδοι της κλάσης και μια σύντομη περιγραφή τους:

void addNode(Node node, RandomAccessFile myFile): Προσθέτει ένα node στο myFile. Το node μετατρέπεται σε 8θέσιο byte[] pair μέσω των μεθόδων ByteBuffer.allocate(4).putInt (…)array() και concatByteArrays. Η τελευταία σελίδα του αρχείου διαβάζεται στον dataPage buffer , εάν είναι γεμάτη δημιουργείται μια νέα η οποία γράφεται στο τέλος του αρχείου, ενώ αν χωράει επιπλέον δεδομένα, γίνονται append στην υπάρχουσα σελίδα και αυτή γράφεται στη θέση απ' όπου διαβάστηκε.

long searchDiskList(Node keyNode, RandomAccessFile myFile): Το προς αναζήτηση node πρώτα μετατρέπεται σε 8θέσιο byte[]. Ακολούθως διαβάζεται το myFile σελίδα-σελίδα, μέχρι να βρεθεί το στοιχείο. Η αναζήτηση του στοιχείου γίνεται στην κεντρική μνήμη ανά 8δες bytes ελέγχοντας εάν το searchPair υπάρχει μέσα στον dataPage buffer. Τέλος επιστρέφεται ο αριθμός προσβάσεων στο δίσκο που απαιτήθηκαν για την αναζήτηση.

static byte[] concatByteArrays(byte[] array1, byte[] array2): Συνενώνει τα array1 και array2 σε ένα τελικό array μήκους array1.length + array2.length (η έμπνευση για τη δημιουργία της βρέθηκε εδώ: https://www.programiz.com/java-programming/examples/concatenate-two-arrays).

Ερώτημα Β.2 : Μέθοδος Κατακερματισμού στον δίσκο.

Παρατήρηση: Για το Β2 ερώτημα μια σελίδα θεωρείται γεμάτη όταν περιέχει 31 και όχι 32 node, τελευταία 8 byte χρησιμοποιούνται για την αποθήκευση της διεύθυνσης της overflow page.

Class PagePointer (org.tuc): Περιέχει δύο μεταβλητές τύπου long. Η μια κρατάει την διεύθυνση της πρώτης σελίδας του αρχείου όπου αποθηκεύονται τα στοιχεία της εκάστοτε *αλυσίδας* ενώ η άλλη κρατά την διεύθυνση της τελευταίας σελίδας της *αλυσίδας*.

Class DiskHash (org.tuc.hash): Διαθέτει όμοιες μεταβλητές με την DiskList αλλά και έναν πίνακα PagePointer[] pagePtr.

void addNode2(Node node, RandomAccessFile myFile): Το στοιχείο node μετατρέπεται σε 8θέσιο byte[] pair. Καλώντας την Hash.hashFunc(int x, int y) υπολογίζεται η θέση pos: pagePtr[pos] από την οποία λαμβάνεται η διεύθυνση της τελευταίας σελίδας της αλυσίδας ώστε να αυτή να μεταφερθεί στην κεντρική μνήμη στον dataPage buffer. Αν η dataPage είναι γεμάτη, προστίθεται μια νέα σελίδα στην αλυσίδα, η οποία γράφεται στο τέλος του αρχείου. Εάν η dataPage χωράει επιπλέον στοιχεία, το node προσαρτιέται σ 'αυτή και η σελίδα ξαναγράφεται στο ίδιο σημείο απ' όπου διαβάστηκε. Εάν pagePtr[pos]=null τότε δημιουργείται μια νέα αλυσίδα και μια νέα σελίδα που γράφεται στο τέλος του αρχείου.

long searchDiskHash(Node keyNode, RandomAccessFile myFile): Μέσω της Hash.hashFunc(int x, int y) υπολογίζεται η θέση pos: pagePtr[pos] που κρατά τη διεύθυνση της *πρώτης* σελίδας της αλυσίδας η οποία μεταφέρεται στην κεντρική μνήμη. Στη συνέχεια εάν το searchPair (x,y) δεν βρίσκεται μέσα στην πρώτη σελίδα, θα μεταφερθούν σειριακά οι σελίδες της αλυσίδας στην μνήμη ,η μια μετά την άλλη, μέχρι να βρεθεί το στοιχείο. Εάν το στοιχείο δε βρεθεί διαβάζονται όλες οι σελίδες της αλυσίδας. Τελικά επιστρέφεται και ο αριθμός των προσβάσεων στον δίσκο που απαιτήθηκαν για την αναζήτηση. Στην περίπτωση που pagePtr[pos]=null (δεν υπάρχει αλυσίδα), η συνάρτηση επιστρέφει 0 εφόσον δεν υπάρχουν σελίδες να διαβαστούν.

byte[] initDataPage(byte[] dataPage): Γεμίζει τον πίνακα που δέχεται ως όρισμα με -1 και τον επιστρέφει.

boolean findPair(byte[] dataPage, byte[] searchPair) : Επιστρέφει true εάν το array searchPair μεγέθους 8bytes περιέχεται μέσα στο array dataPage μεγέθους 256bytes. Διαφορετικά επιστρέφει false.

int getPageIndex(byte[] dataPage): Επιστρέφει την πρώτη θέση του πίνακα dataPage η οποία περιέχει -1, δηλαδή επιστρέφει την πρώτη θέση του πίνακα η οποία είναι 'άδεια' και από αυτή και κάτω μπορούμε να γράψουμε δεδομένα στον πίνακα.

Ο πίνακας PagePointer[] pagePtr βρίσκεται στην κεντρική μνήμη. Τα δεδομένα αυτού του πίνακα θα μπορούσαν να περιέχονται ανά ζευγάρι σε ένα αρχείο, τότε δε θα χρειαζόταν η κλάση PagePointer και το αρχείο θα ήταν χωρισμένο σε Μ σελίδες των 16bytes (8+8bytes για κάθε long). Κάθε σελίδα αυτού του αρχείου αντιπροσωπεύει μια θέση του pagePtr. Συνεπώς θα χρειαζόμασταν μεθόδους που ανάλογα με το αποτέλεσμα της hashFunc θα διάβαζαν/ενημέρωναν την κατάλληλη σελίδα του αρχείου.

ΜΕΡΟΣ Γ : Σύγκριση δομών και τεκμηρίωση

Class Tester (org.tuc.test): Η πιο βασική μέθοδος της κλάσης είναι η void doTest() η οποία καλείται στην main. Οι βασικότερες μεταβλητές της αποθηκεύουν τα αποτελέσματα του πειράματος (πχ. meanComparisonsA1, meanAccessesA1..)

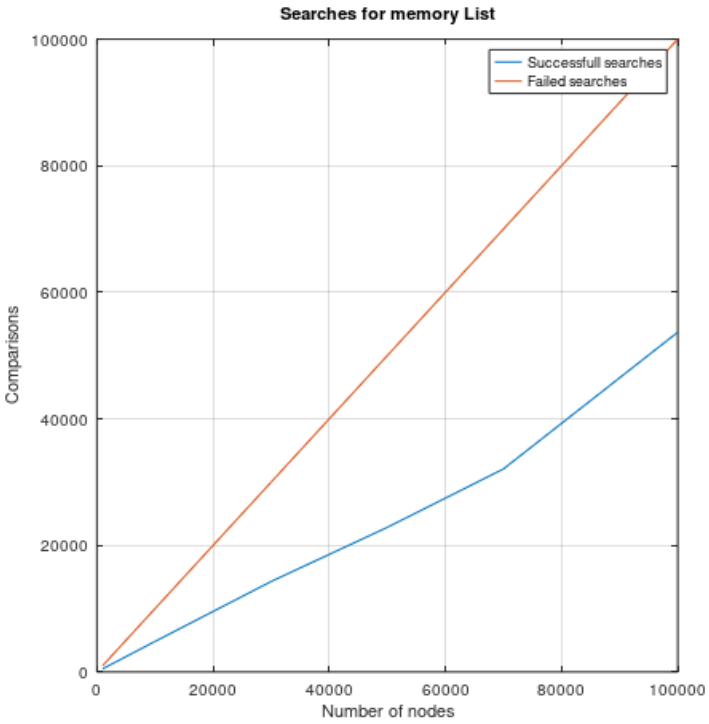
Node[] getRandomNodes(int numberOfNodes) : Επιστρέφει #numberOfNodes στο πλήθος τυχαία στοιχεία (x,y) που παίρνουν τιμές στο διάστημα minIntNumber<=x,y<=maxIntNumber. Στην άσκηση μας είναι minIntNumber=0 και maxIntNumber=2^18.

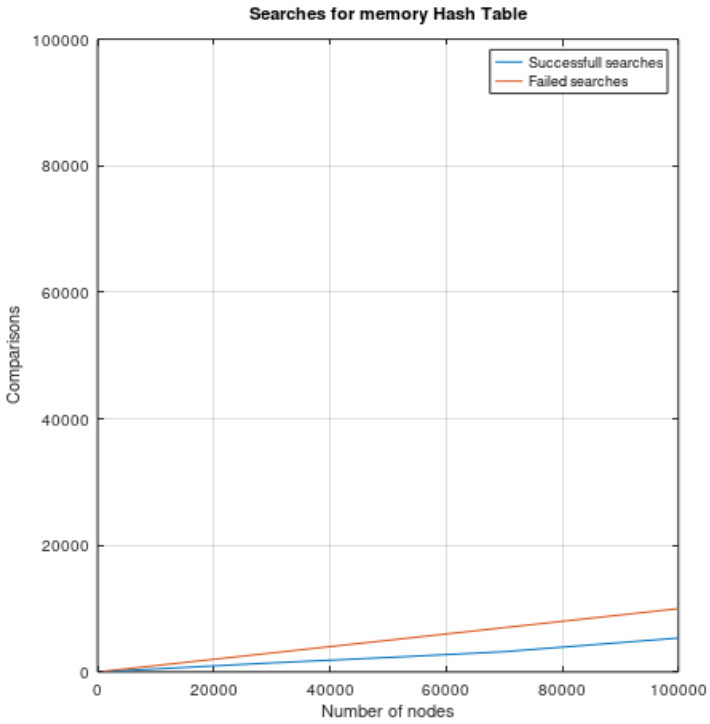
public Node[] getRandomNodes(int numberOfNodes, int min , int max): Ακριβώς ίδια με την παραπάνω με τη διαφορά ότι κατά την κλήση της μπορούν να καθοριστούν τα όρια τιμών για τα x και y. (Χρησιμοποιείται για την παραγωγή nodes που δεν υπάρχουν στις δομές για τις αποτυχημένες αναζητήσεις…) .

void doTest(): Κατασκευάζει καθεμιά από τις 4 δομές για αριθμό στοιχείων K = 1.000, 10.000, 30.000, 50.000, 70.000, 100.000 ,εκτελεί 100 επιτυχείς αναζητήσεις και 100 αποτυχημένες για κάθε τιμή του K σε κάθε δομή. Επίσης αποθηκεύει για κάθε αναζήτηση του μέρους Α τον μέσο αριθμό συγκρίσεων που απαιτήθηκαν ενώ για κάθε αναζήτηση του Β μέρους αποθηκεύει τον μέσο αριθμό των προσβάσεων στον δίσκο.

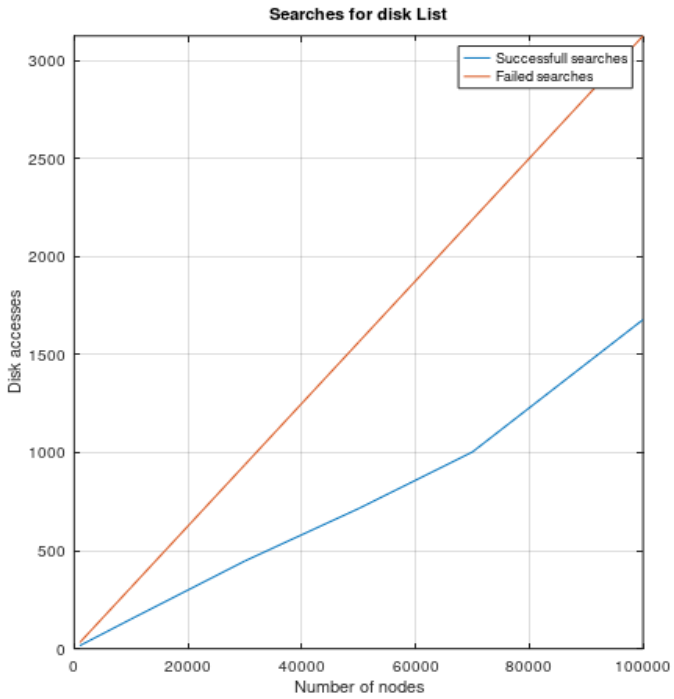
void meanComparisonsA1(), void failComparisonsA1(), void meanComparisonsA2(), void failComparisonsA2(), void printMeanAccessesB1(), void printFailAccessesB1(), void prinMeanAccessesB2(), void printFailAccessesB2() : εκτυπώνουν τα αποτελέσματα στην οθόνη.

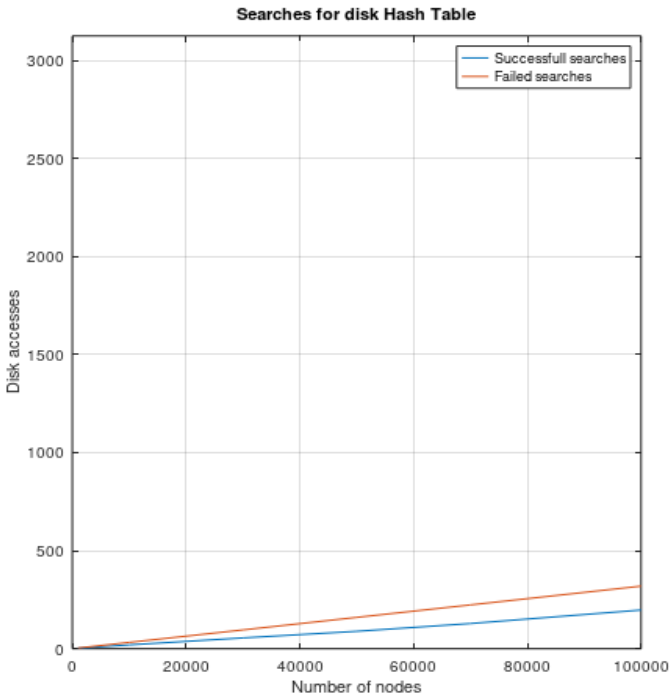
Διαγράμματα | ΜΈΡΟΣ Α:





Διαγράμματα | ΜΈΡΟΣ Β:





Ο πίνακας κατακερματισμού είναι εμφανώς πιο αποδοτική δομή όσο αφορά την αναζήτηση στοιχείων και αυτό οφείλεται στο hashing. Λόγω του hashing δεν είναι απαραίτητη η διάσχιση όλων των στοιχείων. Τα μόνα στοιχεία που διασχίζονται είναι αυτά που «ανήκουν» στις λίστες/θέσεις του πίνακα όπου υποδεικνύει το hashing που έγινε κατά την προσθήκη τους. Ουσιαστικά ο πίνακας κατακερματισμού ψάχνει κάθε στοιχείο σε μια «λίστα-υποσύνολο». Πιο συγκεκριμένα σύμφωνα με τα αποτελέσματα του πειράματος και τα παραπάνω διαγράμματα -για παράδειγμα- αναζητώντας με επιτυχία 100 στοιχεία από τα 100.000 στην λίστα μνήμης, απαιτούνται περίπου 50.000 συγκρίσεις, ενώ για την ίδια αναζήτηση στον πίνακα κατακερματισμού της μνήμης απαιτούνται σχεδόν 5.000 , δηλαδή απαιτείται σχεδόν το ένα δέκατο των συγκρίσεων. Όμοια και για το δίσκο, κατά την επιτυχή αναζήτηση των 100 στοιχείων από τα 1.000 στην λίστα απαιτούνται περίπου 15 προσβάσεις στον δίσκο ενώ για αναζήτηση στον πίνακα κατακερματισμού απαιτούνται γύρω στις 2 προσβάσεις στο δίσκο. Ανάλογα συμπεράσματα ισχύουν και για τις αποτυχημένες αναζητήσεις.

Πηγές πληροφόρησης-έμπνευσης:
https://www.programiz.com/java-programming/examples/concatenate-two-arrays
https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/