Αναφορά Κώδικα 3ης Εργαστηριακής Άσκησης

ΠΛΗ202

Λεοντής Παναγιώτης ΑΜ 2018030099

Γενική Εικόνα:

Αρχικά για να τρέξει το πρόγραμμα πληκτρολογείτε το path του αρχείου ως argument. Υπάρχει η κλάση *Console* όπου περιέχεται η main(). Έπειτα στο πακέτο BinSearchTree υπάρχουν οι κλάσεις *Node* και *BinarySearchTree* για την υλοποίηση του δυαδικού δέντρου έρευνας. Συγκεκριμένα χρησιμοποίησα την υλοποίησή μου για την προηγούμενης άσκηση καθώς οι ζητούμενες μέθοδοι ήταν οι ίδιες. Έπειτα έχουμε τις κλάσεις *LinearHashing* και *Bucket* για να δημιουργήσουμε τον πίνακα κερματισμού όπως ζητήθηκε. Στην κλάση console δημιούργησα τα 3 αντικείμενα lHashing_05, lHashing_08 και bTree τα οποία θα ήταν οι δύο πίνακες με split rule u>0.5 και u>0.8 αντίστοιχα και το δυαδικό δέντρο. Μέσα στην λίστα intList περνάω όλους τους αριθμούς του αρχείου αφού το διαβάσω στην μέθοδο createList() με ένα *RandomAccessFile*. Δημιουργώ ένα *LinearHashing* αντικείμενο deletetNums για να γνωρίζω ποιοι αριθμοί έχουν ήδη επιλεχθεί για αναζήτηση και διαγραφή.

Κλάση LinearHashing:

- loadFactor() Υπολογίζει και επιστρέφει το αποτέλεσμα των συνολικών κλειδιών προς το συνολικό διαθέσιμο χώρο στο πίνακα.
- hashFunction() Επιστρέφει τον αριθμό του κουβά που πρέπει να τοποθετηθεί το κλειδί. Υπολογίζει το key%M όπου M ο αρχικός αριθμός κουβάδων. Αν ο κουβάς που προορίζεται το κλειδί προηγείται του επόμενου κουβά για split χρησιμοποιώ key%2M (δευτερη hash function).
- bucketSplit() Δημιουργώ μια παραπάνω θέση στο πεδίο για τα buckets καλώ την split της *Bucket* ώστε να διασπαστεί ο κουβάς και ενημερώνω την κλάση LinearHashig(availSpace,numOfBuckets). Επιπλέον κάνω τον κατάλληλο έλεγχο για το αν έχουμε αλλαγή φάσης. Αν ναι ορίζω τον επόμενο κουβά για split στη θέση 0 και διπλασιάζω το M. Αλλιώς απλά μεταφέρω τον επόμενο κουβά για split μια θέση δεξιά
- bucketMerge() δημιουργώ ένα νέο πεδίο από Buckets με μια θέση λιγότερη και αντιγράφω όλους τους κουβάδες εκτός από τον τελευταίο. Κάνω έλεγχο για το εάν πρέπει να γίνει αλλαγή φάσης. Εάν ναι, υποδιπλασιάζω το M και θέτω τον κουβά για διάσπαση αυτόν στην θέση M-1. Αλλιώς απλά μεταφέρω τον επόμενο κουβά για split μία θέση αριστερά. Ενημερώνω την κλάση LinearHashing(availSpace,numOfBuckets), καλώ την merge της *Bucket* και έπειτα ενημερώνω και το πεδίο με τους κουβάδες της *LinearHashing*

- insertKey() Καλώ την μέθοδο insert της *Bucket* και την hashFunction() για να υπολγίσω σε ποια θέση θα πάει το κλειδί. Έπειτα κάνω έλεγχο με το loadFactor για το αν χρειάζεται split.
- deleteKey() Αντίστοιχα καλώ την μέθοδο delete της *Bucket* και την hashFunction() για να υπολογίσω σε ποια θέση είναι το κλειδί για διαγραφή. Έπειτα κάνω έλεγχο με το loadFactor() για το αν χρειάζεται split ή merge.
- searchKey() Καλώ την search της *Bucket* και την hashFunction() για να υπολογίσω σε ποια θέση να ψάξω για το κλειδί

Κλάση Bucket:

- split() Αρχικά ελέγχω για το αν κάθε κλειδί πρέπει να μεταφερθεί σε διαφορετικό κουβά. Εάν ναι κάνω insert στον νέο κουβά και ενημερώνω την κλάση LinearHashing αφαιρώντας 1 από το numOfKeys καθώς κατά το insert αυξάνεται κατά 1 ενώ στην πραγματικότητα δεν προστίθεται κάποιο κλειδί. Στην συνέχεια ελέγχω για το αν υπάρχει overflowBucket και αν ναι τον κάνω split. Αν αδείασει τον κάνω free και ενημερώνω την *LinearHashing*.
- insert() Αρχικά ψάχνω εάν το κλειδί υπάρχει ήδη, εάν ναι κάνω σταματάω. Αν όχι τσεκάρω εάν ο κουβάς είναι άδειος και το τοποθετώ μέσα στον κουβά. Αν είναι γεμάτος θα χρησιμοποιήσω το overflowBucket. Εάν δεν υπάρχει τότε το δημιουργώ. Έπειτα καλώ ξανά την insert της Bucket για το overflowBucket.
- search() Διασχίζω το πεδίο με τα κλειδιά που περιέχει ο κουβάς και αν βρεθεί το κλειδί επιστρέφω true. Εάν δεν βρεθεί τσεκάρω και στο overflowBucket καλώντας την search για αυτό. Εάν δεν βρεθεί το κλειδί επιστρέφω false.
- merge() Μέσα σε while μέχρι να αδειάσει ο κουβάς που κάνω merge καλώ την insert και το κλειδί που παίρνει ως όρισμα το επιστρέφει κάθε φορά η μέθοδος removeLastKey(). Αυτή αφαιρεί το τελευταίο κλειδί από τον κουβά ή από το overflowBucket εάν υπάρχει και επιστρέφει την τιμή του.
- delete() Αρχικά ψάχνω μέσα στον κουβά το κλειδί για διαγραφή. Αν το βρω τότε ελέγχω για το εάν υπάρχει overflowBucket και κάνω insert το τελυταίο του κλειδί στον κουβά μέσω της removeLastKey(). Αν δεν βρω το κλειδί στον κουβά ελέγχω το overflowBucket. Εάν είναι άδειος τότε τον κάνω free.

Υλοποίηση Μετρήσεων:

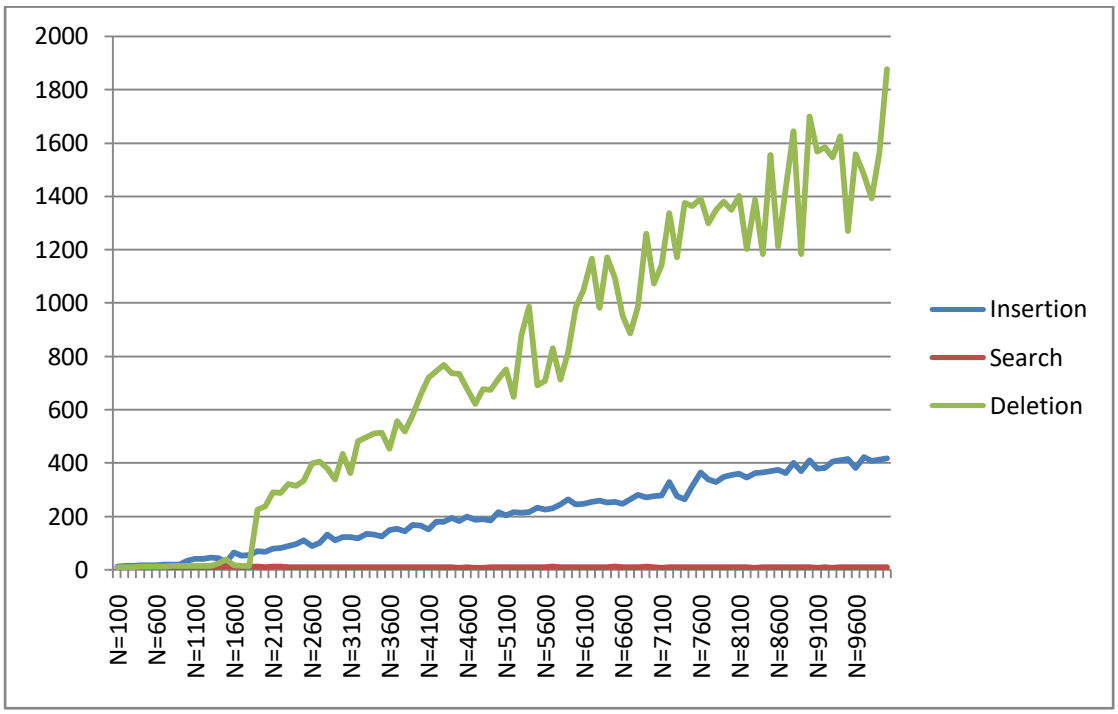
Αρχικά οι κλάσεις *BinarySearchTree* και *LinearHashing* έχουν ως member variable μια μεταβλητή compares. Αυτή αυξάνεται κάθε φορά που έχουμε συνθήκη (if,while, for) ή κάποια ανάθεση μέσω των μεθόδων increaseCompares() και increaseComparesBy() της κάθε κλάσης. Στην μέθοδο printResults της *Console* εκτυπώνω τον πίνακα και έπειτα καλώ τις μεθόδους insert100(), search50(), delete50() μέσα σε μία for ώσπου να φτάσω το συνολικό μέγεθος του αρχείου.

- insert100() Ξεκινώ να διαβάζω το αρχείο μου από την τρέχουσα θέση του pointer στο αρχείο(την πρώτη φορά βρίσκεται στην αρχή) για 100 φορές και κάθε φορά κάνω εισαγωγή του αριθμού στα 3 αντικείμενα μου. Στην συνέχεια δημιουργώ ένα νέο πεδίο με τυχαία επιλεγμένους 50 από τους συνολικούς αριθμούς που έχουν εισαχθεί. Αυτά γίνονται στην μέθοδο pickRandomNums(). Σε αυτή ο pointer του αρχείου βρίσκεται στο σημείο όπου σταμάτησε στην insert100(), διαιρώ με το 4 και βρίσκω πόσους αριθμούς έχει διαβάσει. Μέσα σε μια while παίρνω αριθμούς τυχαία μέσω της Math.Random από την intList, ελέγχω εάν έχουν ξαναεπιλεχθεί και αν όχι τους προσθέτω στο πεδίο με τους αριθμούς για αναζήτηση (searchNums) και στο LinearHashing deletedNums() για να γνωρίζω ότι έχει επιλεχθεί προηγουμένως.
- search50() Μέσα σε μία for για 50 επαναλήψεις παίρνω κάθε φορά έναν αριθμό από το πεδίο που παράγεται στην pickRandomNums(), το searchNums και τον αναζητώ στα 3 αντικείμενα.
- delete50() Αντίστοιχα μέσα σε μια for 50 επαναλήψεων παίρνω έναν αριθμό από το searchNums() κάθε φορά και κάνω διαγραφή και για τα 3 αντικείμενα.

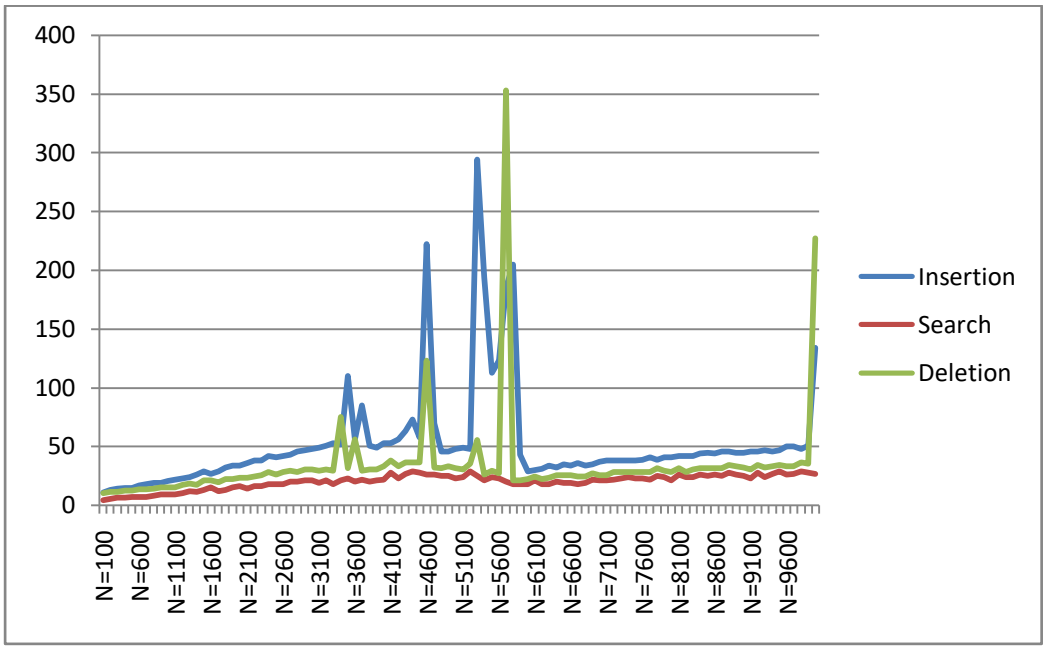
Στο τέλος και των τριών μεθόδων insert100(), search50(), delete50() καλώ την μέθοδο eraseCompares() η οποία μηδενίζει τα compares και των 3 αντικειμένων για να μην οδηγηθούμε σε λάθος αποτελέσματα.

Μετρήσεις:

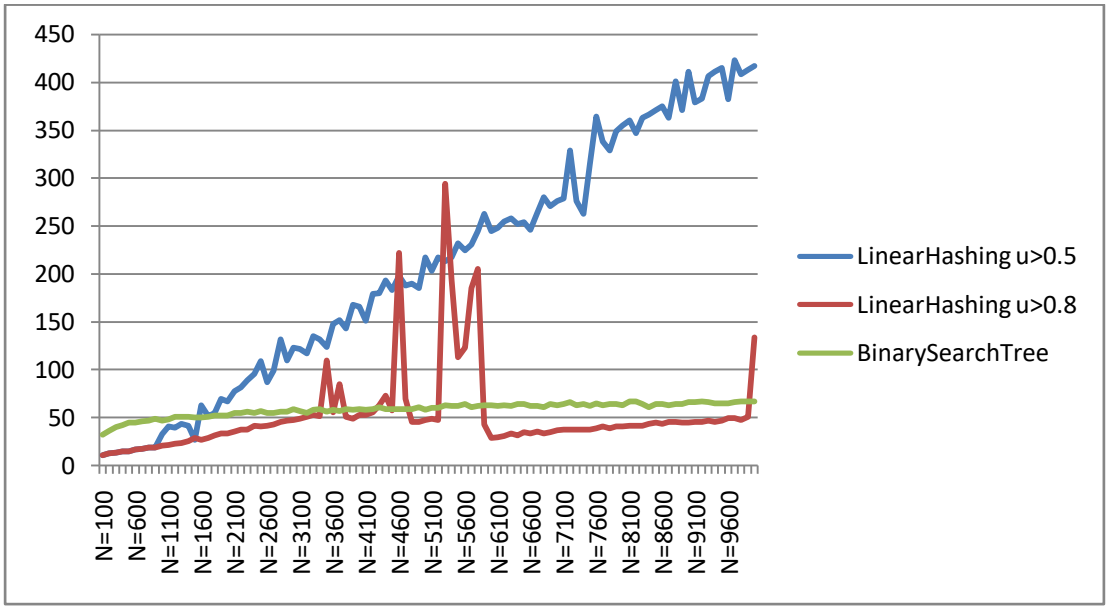
Παρατίθεται το διάγραμμα του μέσου όρου συγκρίσεων για mergeRule<0.5 και splitRule>0.5 για τις πράξεις εισαγωγής αναζήτησης και διαγραφής.



Παρατίθεται το διάγραμμα μέσου όρους συγκρίσεων για mergeRule<0.5 και splitRule>0.8 για τις πράξεις εισαγωγής αναζήτησης και διαγραφής.



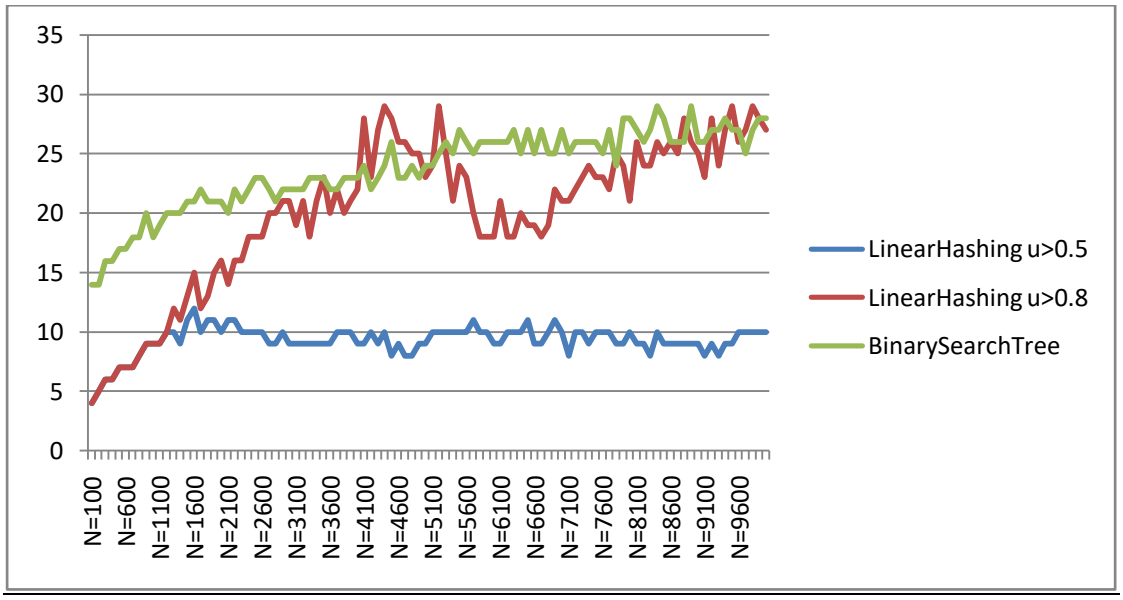
Παρατίθεται το διάγραμμα του μέσου αριθμού συγκρίσεων στις 3 υλοποιήσεις για την εισαγωγή



Παρατηρούμε ότι τον μεγαλύτερο μέσο όρο συγκρίσεων ανά εισαγωγή τον εμφανίζει η δομή hash table κατά την οποία ισχύει ότι split rule factor > 50% και merge rule factor < 50%. Αυτό συμβαίνει διότι γίνονται πιο νωρίς τα split (έπειτα από κάθε εισαγωγή) και επίσης, όσο αυξάνεται ο αριθμός των κλειδιών που έχουν

εισαχθεί, κάθε split χρειάζεται περισσότερες πράξεις για να πραγματοποιηθεί. Το load factor κινείται γύρω από το 0.5(με κάποιες εισαγωγές το υπερβαίνει και με κάποιες διαγραφές είναι ελάχιστα μικρότερο). Επομένως έχουμε συνέχεια splits και merges. Για το δεύτερο hash table τα split γίνονται με μικρότερο ρυθμό έπειτα από εισαγωγή. Αυτό διότι το split rule(0.8) απέχει από το merge rule(0.5). Οι εισαγωγές σε Δ.Δ.Ε. ακολουθούν λογαριθμική πολυπλοκότητα, η οποία όσο αυξάνεται η τιμή του μεγέθους της εισόδου δίνει σχεδόν σταθερό αριθμό συγκρίσεων ανά εισαγωγή. Παρατηρούμε ότι στην αρχή, για μικρό μέγεθος εισόδου, η απόδοση των δύο διαφορετικών δομών hash table ταυτίζεται. Αυτό συμβαίνει διότι τότε είναι που αρχίζει να γεμίζει η κάθε δομή με κλειδιά και αφού δεν έχουν πραγματοποιηθεί ακόμα splits κατά τις εισαγωγές, δεν πραγματοποιούνται ούτε merges κατά τις διαγραφές.

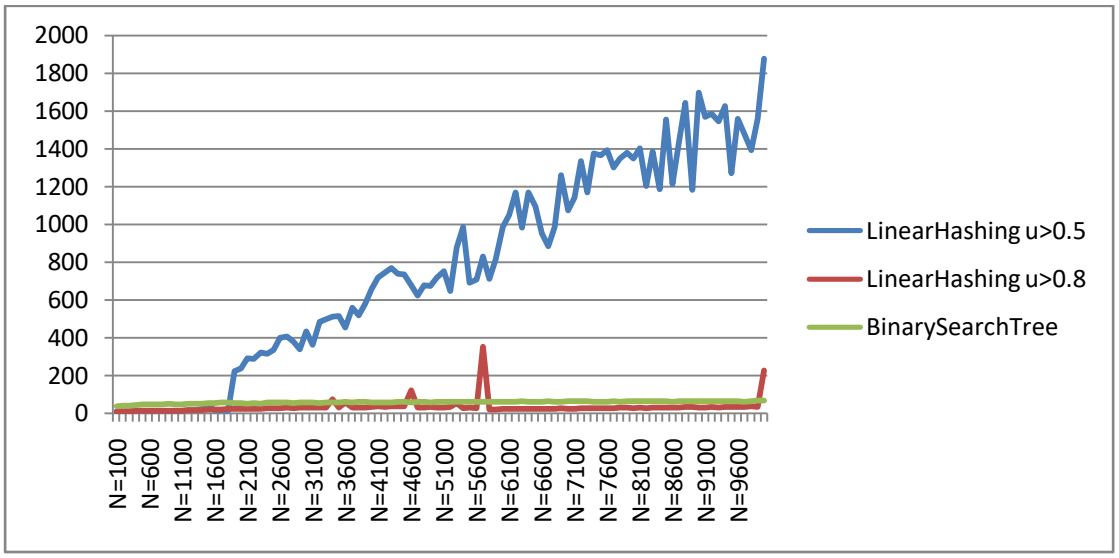
Παρατίθεται το διάγραμμα του μέσου αριθμού συγκρίσεων στις 3 υλοποιήσεις για την αναζήτηση



Παρατηρούμε ότι τον μεγαλύτερο μέσο όρο συγκρίσεων ανά αναζήτηση τον εμφανίζε κυρίως η δομή Binary Search Tree. Αυτό συμβαίνει διότι η αναζήτηση στο Δ.Δ.Ε. έχει πολυπλοκότητα O(log(N)). Στη συνέχεια, μικρότερο μέσο αριθμό συγκρίσεων ανά αναζήτηση, σε κάθε τιμή του μεγέθους της εισόδου, παρουσιάζει η δομή hash table κατά την οποία ισχύει ότι split rule factor > 0.8 και merge rule factor < 0.5. Σε αυτό έχουμε λιγότερα splits οπότε τα Buckets περιέχουν μεγαλύτερο πλήθος κλειδιών από το hash table με split rule>0.5. Στην αυτή την περίπτωση(split rule>0.5) έχουμε περισσότερα splits. Άρα τα Buckets είναι λιγότερο γεμάτα άρα απαιτούνται λιγότερες πράξεις. Αναμενόμενο τα hash tables να έχουν μικρότερο μέσο όρο συγκρίσεων διότι απλώς διασχίζουμε σειριακά το Bucket για την εύρεση

του κλειδιού. Για μικρό μέγεθος εισόδου η απόδοση των δυο hash table ταυτίζεται καθώς δεν έχουν γίνει τα πρώτα splits κατά την εισαγωγή.

Παρατίθεται το διάγραμμα του μέσου αριθμού συγκρίσεων στις 3 υλοποιήσεις για την διαγραφή



Παρατηρούμε ότι τον μεγαλύτερο μέσο όρο συγκρίσεων ανά διαγραφή τον εμφανίζει η δομή hash table κατά την οποία ισχύει ότι split rule factor > 0.5 και merge rule factor < 0.5. Αυτό διότι μετά τις διαγραφές πραγματοποιούνται και πολλά merges. Έπεται το Binary Search Tree. Παρατηρούμε το hash table με split rule>0.8 να έχει το μικρότερο μέσο όρο συγκρίσεων για σχεδόν όλα τα μεγέθη εισόδου. Αυτό διότι μετά από διαγραφή πιθανόν να μην χρειαστεί κάποιο merge καθώς ο load factor έχει περιθώριο από το 0.8 έως το 0.5. Παρατηρούμε πως για μικρό μέγεθος εισόδου η απόδοση των δυο hash table ταυτίζεται. Αυτό διότι τότε ουσιαστικά αρχίζει να γεμίζει η κάθε δομή. Αν σκεφτούμε ότι δεν έχουν γίνει split στην εισαγωγή, τότε δεν γίνονται ούτε merges ούτε διαγραφές.