Project: Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα
“N-grams detection”

Ομάδα:

Αντωνία Αθανασάκου 1115201400004

Στεφανία Πάτσου 1115201400156

Περιεχόμενα

N-grams detection: Part 1

Δομές

Στο πρώτο μέρος της εργασίας κληθήκαμε να δημιουργήσουμε τις δομές στις οποίες θα αποθηκεύονται τα N-grams και τις συναρτήσεις εισαγωγής, διαγραφής και εύρεσης. Δημιουργήσαμε, λοιπόν, ένα δείκτη σε δομή “arrayOfStructs” ο οποίος αποτελούταν από ένα πίνακα από δομές “dataNode”, το μέγεθος του πίνακα (στην αρχή ήταν 10 και διπλασιαζόταν κάθε φορά που δεν χωρούσε κάποιο στοιχείο), καθώς και τη θέση του στοιχείου που επρόκειτο να εισαχθεί κάποιο καινούριο N-gram. Μια δομή “dataNode” αποτελείται από έναν δείκτη σε πίνακα που αποθηκεύει τις επόμενες λέξεις, έναν δυναμικό πίνακα από χαρακτήρες για την συγκεκριμένη λέξη και μια λογική μεταβλητή που προσδιορίζει αν η λέξη είναι final (τελευταία του N-gram) ή όχι.

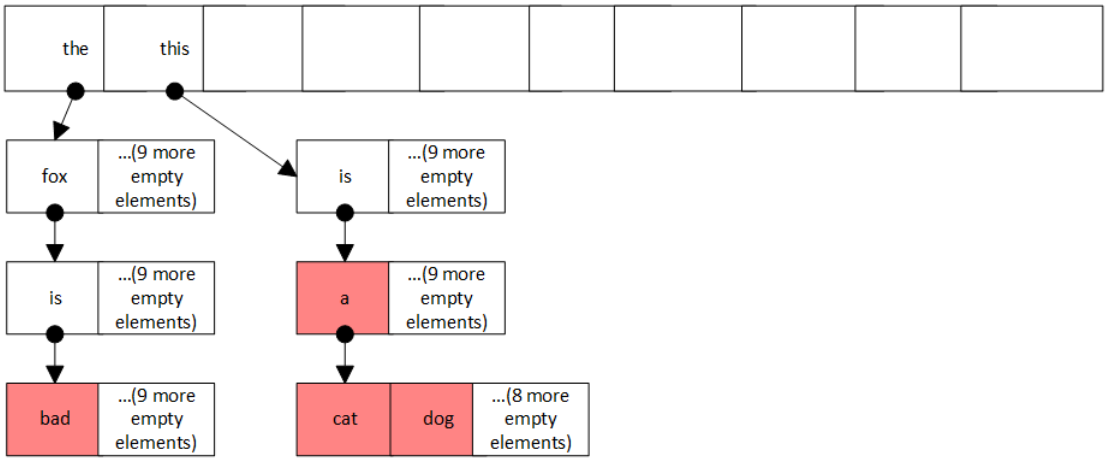
Στην συνάρτηση main, διαβάζαμε αρχικά τις γραμμές του αρχείου init_file και εισαγάγαμε τα N-grams στον πίνακα. Κατά την εισαγωγή το κάθε στοιχείο ταξινομούταν απευθείας με τη βοήθεια της συνάρτησης insertionSort. Έπειτα διαβάζαμε από το αρχείο query_file, και ανάλογα με το πρώτο γράμμα της κάθε γραμμής (A, D, Q) εκτελούσαμε εισαγωγή, διαγραφή ή αναζήτηση. Κατά την αναζήτηση δημιουργούσαμε έναν πίνακα από λέξεις, κάθε φορά που έπρεπε να εκτελέσουμε την αναζήτηση ξεκινώντας από διαφορετική λέξη κάθε φορά (γεγονός που ήταν πολύ χρονοβόρο και αργότερα το καταργήσαμε).

Για την αποφυγή των διπλότυπων N-grams σε ένα ερώτημα Q, δημιουργήσαμε έναν πίνακα στον οποίο αποθηκεύαμε τα εκτυπωμένα N-grams και με βάση τον οποίον ελέγχαμε αν η πρόταση έχει ήδη εκτυπωθεί ή όχι.

Παρακάτω φαίνεται η διαμόρφωση του αρχικού πίνακα μετά την εισαγωγή των φράσεων:

- this is a cat
- this is a
- this is a dog
- the fox is bad

Οι χρωματισμένοι κόμβοι είναι final.



Εικόνα 1: Δομή πίνακα μετά από εισαγωγές

Για τη διαγραφή των n-grams χρησιμοποιήσαμε μια δομή στοίβας που αποτελούταν από έναν πίνακα από ακεραίους, κάθε κόμβος του οποίου ήταν ο αριθμός του στοιχείου που έπρεπε να διαγραφεί, το μέγεθος του πίνακα και έναν ακέραιο που έδειχνε τη θέση του τελευταίου στοιχείου του πίνακα.

Αρχεία δομών και συναρτήσεων

Πέρα από την main.c , υλοποιήσαμε και τα εξής αρχεία:

- stack.c : Περιέχει τις μεθόδους για την διαχείριση μίας στοίβας, όπως αρχικοποίηση (initializeStack), διπλασιασμός (doubleLengthOfStack), διαγραφή (deleteStack), διαγραφή ενός στοιχείου (pop), εισαγωγή (push), απεικόνιση (displayStack), έλεγχος για το αν είναι κενή (isEmpty).
- struct.c : Περιέχει τις μεθόδους για την διαχείριση της ολικής δομής μας, όπως αρχικοποίηση (initializeArray), διπλασιασμός (doubleLength), διαγραφή (deleteArray) , διαγραφή ενός στοιχείου (deleteDataNode).
- test.c : Περιέχει τα unit testing για την κάθε συνάρτηση.
- auxMethods.c : Περιέχει τις μεθόδους που μας βοηθούν να υλοποιήσουμε τις μεθόδους στο func.c, όπως binarySearch , insertionSort κτλπ.
- func.c : Περιέχει τις κύριες μεθόδους για την υλοποίηση μας δηλαδή τις: εισαγωγή ngram ενός ngram(insert_ngram) , αναζήτηση ενός ngram (search_ngram) διαγραφή ενός ngram (delete_ngram).

Unit Testing

Προκειμένου να ελέγξουμε τη λειτουργικότητα των συναρτήσεών μας, υλοποιήσαμε μόνες μας στο αρχείο “test.c” κάποιες συναρτήσεις που καλούσαν τις ήδη υπάρχουσες συναρτήσεις μας και έλεγχαν αν τα αποτελέσματά τους ήταν τα επιθυμητά. Η συνάρτηση main καλούσε μια συνάρτηση “testAllFunctions” και αυτή με τη σειρά της καλούσε όλες τις υπόλοιπες συναρτήσεις. Συγκεκριμένα οι συναρτήσεις που υλοποιήσαμε είναι:

- void test_initializeArray();
- void test_doubleLength();
- void test_deleteArray();
- void test_deleteDataNode();
- void test_insert_ngram();
- void test_search_ngram();
- void test_binarySearch();
- void test_insertionSort();
- void test_callBasicFuncs();
- void testAllFunctions();
- void test_initializeStack();
- void test_doubleLengthOfStack();
- void test_push();
- void test_isEmpty();
- void test_pop();
- void test_deleteStack();

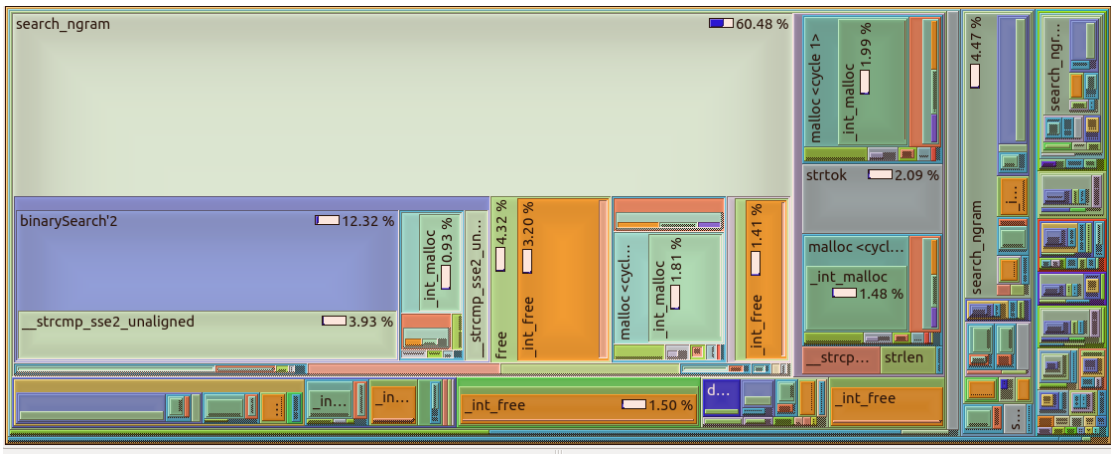
- void test_delete_ngram();
- void test_checkIfStringExists();
- void test_deleteArrayOfWords();
- void test_stringToArray();
- void test_initialize();
- void test_deletionSort();
- void test_executeQueryFile();

Στατιστικά εκτέλεσης

ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ	ΧΡΟΝΟΙ ΕΚΤΕΛΕΣΗΣ	ΟΡΘΟΤΗΤΑ ΑΠΟΤΕΛΕΣΜΑΤΩΝ (✓)
test	0.005 sec	X

Χρονικά διαγράμματα

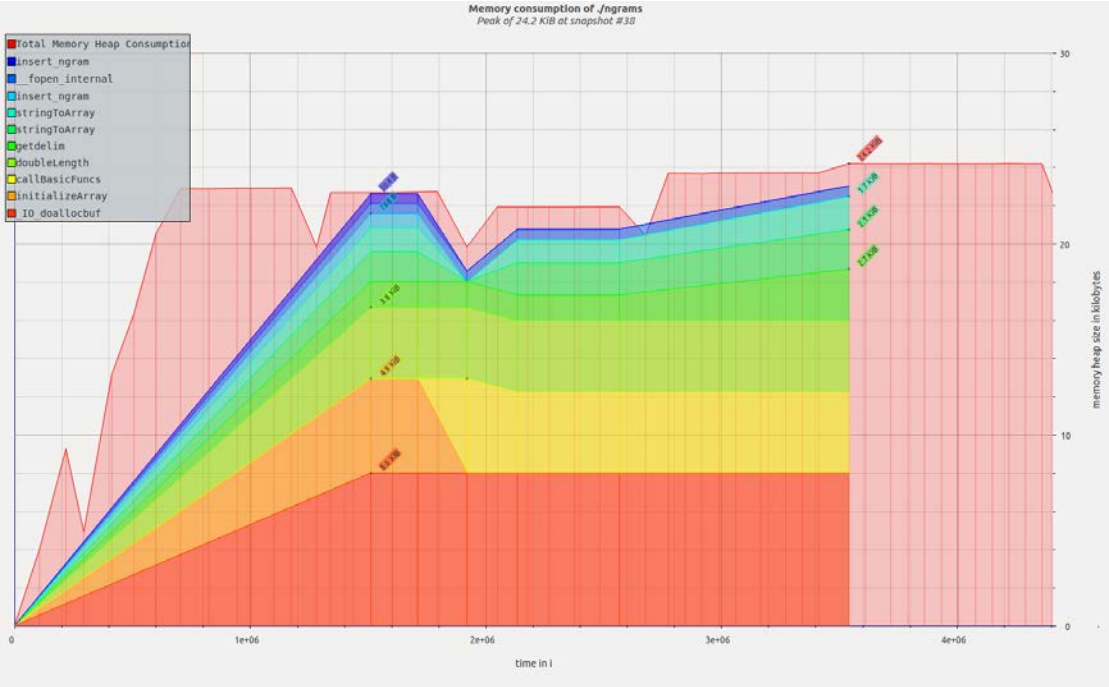
Παρακάτω φαίνονται πληροφορίες για τα χρονικά ποσοστά που καταλαμβάνει η κάθε συνάρτηση.



Εικόνα 2: test

Χωρικά διαγράμματα

Παρακάτω παρουσιάζεται η κατανομή του χώρου όσων αφορά τις συναρτήσεις.



Εικόνα 3: test

Σχόλια Εξεταστή

Δημιουργήσαμε διάφορα init_files και query_files, τα οποία δούλευαν σωστά με το πρόγραμμά μας. Όμως τα αρχεία test και small έβγαζαν λάθη. Κατά την προφορική εξέταση μας συμβούλευσε ο υπεύθυνος καθηγητής να αλλάξουμε τον τύπο μιας μεταβλητής (read) με την οποία διαβάζαμε γραμμές από το query_file και να την ορίσουμε ως ακέραια. Αρχικά την είχαμε ορίσει ως char με αποτέλεσμα στο byte 255 να θεωρεί ότι υπάρχει EOF. Επιπλέον έπρεπε να αλλάξουμε τον τρόπο που εκτυπώναμε τα αποτελέσματα και να αφαιρέσουμε τις περιττές κενές γραμμές στην εκτύπωση. Οι συναρτήσεις για unit testing έπρεπε να καλούνται σε διαφορετικό αρχείο με διαφορετική main. Επίσης μας πρότειναν να υλοποιήσουμε τις συναρτήσεις αυτές με τη βοήθεια κάποιου ήδη έτοιμου framework. Επιπροσθέτως οι λέξη σε κάθε κόμβο dataNode έπρεπε να αποθηκεύεται στατικά και αν ξεπερνούσε κάποιο προκαθορισμένο αριθμό χαρακτήρων (20 χαρακτήρες) να αποθηκεύεται δυναμικά. Τέλος η συνάρτηση insertionSort έπρεπε να χρησιμοποιεί τη συνάρτηση memmove, αντί για κάποια επαναληπτική δομή και ανάθεση.

Διορθώσεις

Αφού διορθώσαμε τις συναρτήσεις και τις δομές μας, χρησιμοποιήσαμε το framework CuTest. Καθεμία από εμάς, υλοποίησε έναν αριθμό από Test Functions για το κάθε αρχείο. Test units από το 1ο μέρος της άσκησης έχουν τροποποιηθεί ανάλογα με το framework. Πολλά από αυτά δεν έχουν αλλάξει ως προς τις περιπτώσεις ελέγχου. Τα αρχεία αυτά βρίσκονται στον φάκελο cutest-1.5 με ξεχωριστό Makefile από αυτό της λειτουργίας της άσκησης.

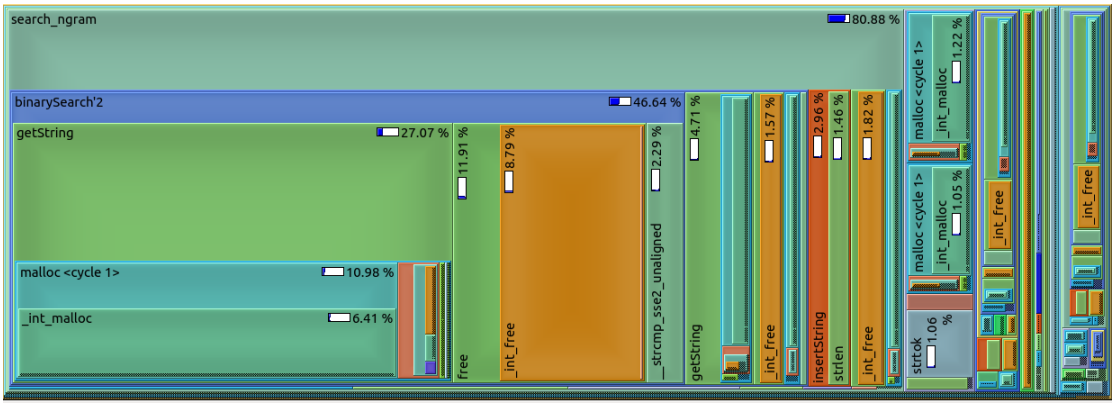
Στη συνέχεια τρέξαμε τα αρχεία test και small και τα αποτελέσματα ήταν σωστά.

Στατιστικά εκτέλεσης

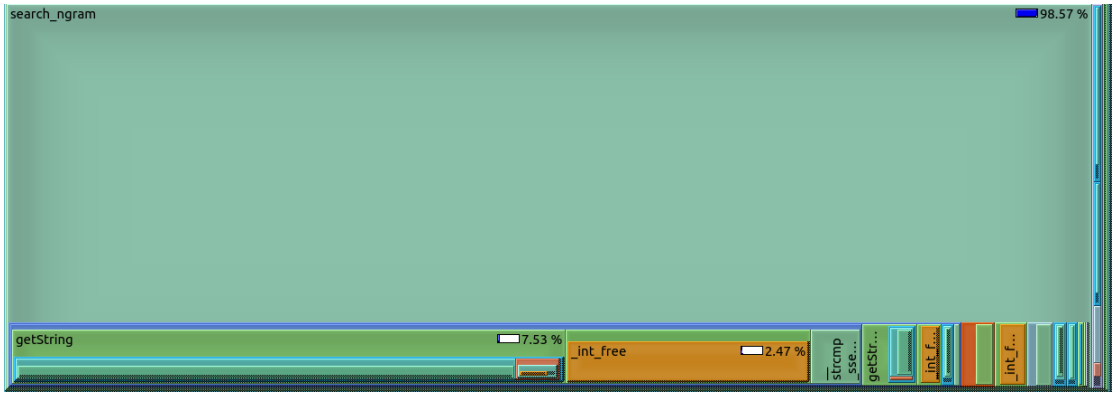
ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ	ΧΡΟΝΟΙ ΕΚΤΕΛΕΣΗΣ	ΟΡΘΟΤΗΤΑ ΑΠΟΤΕΛΕΣΜΑΤΩΝ (✓)
test	0.004 sec	✓
small	38.3 sec	✓

Χρονικά διαγράμματα

Παρακάτω παρουσιάζονται διαγράμματα που δείχνουν τη χρονική διάρκεια των συναρτήσεων κατά την εκτέλεση των αρχείων εισόδου.



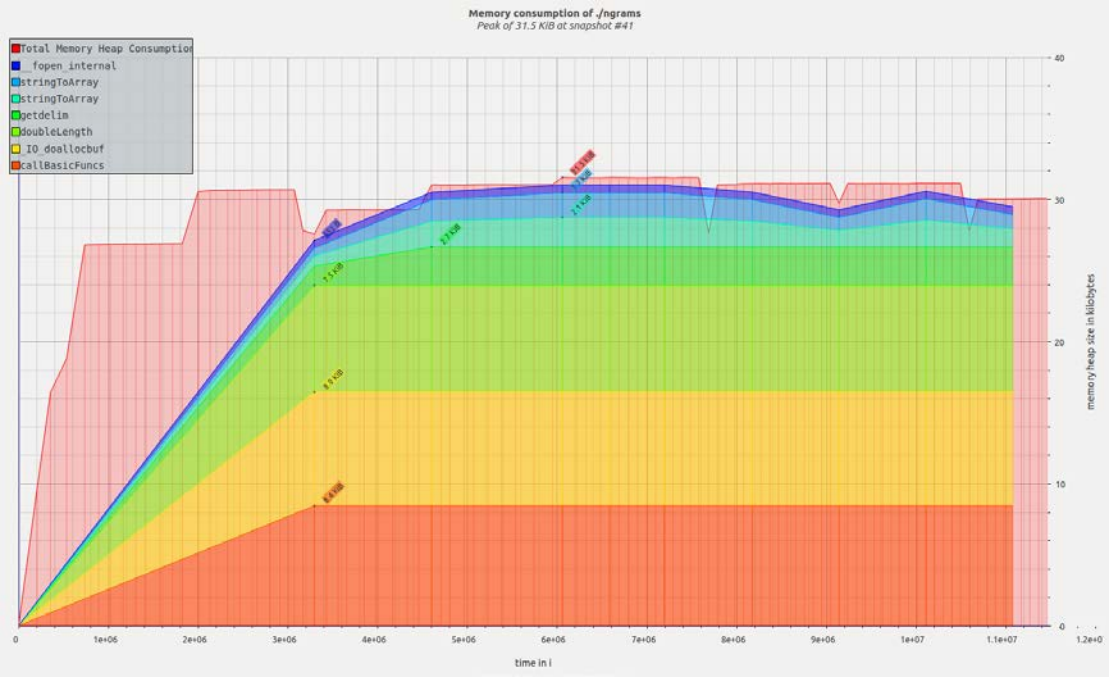
Εικόνα 4: test



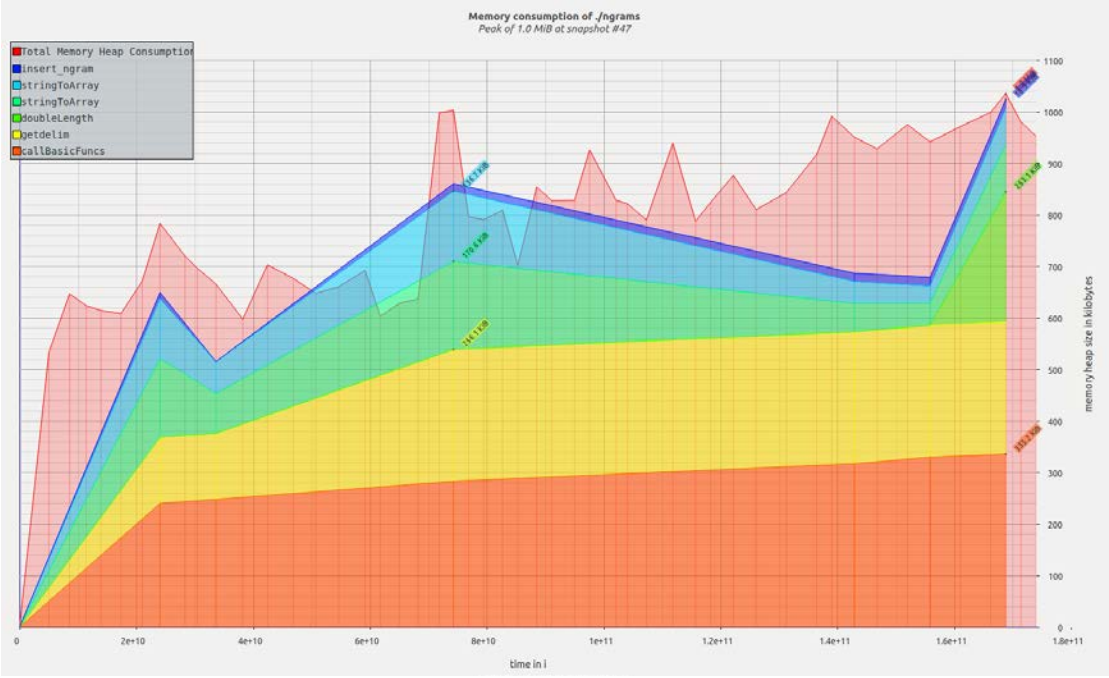
Εικόνα 5: small

Χωρικά διαγράμματα

Τα παρακάτω διαγράμματα αντιπροσωπεύουν τη μνήμη που καταλάμβανε κάθε συνάρτηση, για κάθε αρχείο εισόδου.



Εικόνα 6: test



Εικόνα 7: small

N-grams detection: Part 2

Δομές

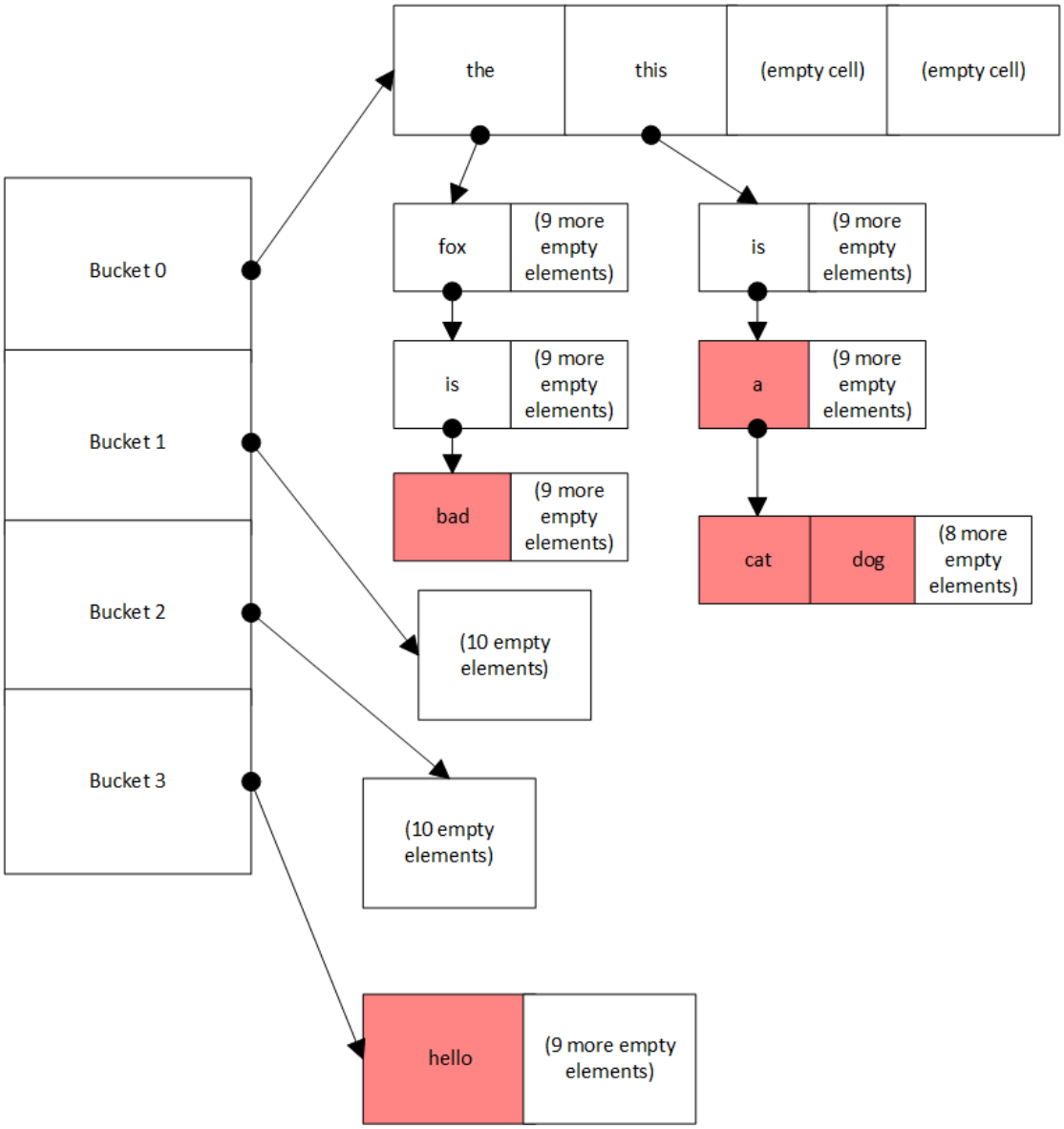
Πίνακας Κατακερματισμού

Στο δεύτερο μέρος της εργασίας, υλοποιήσαμε έναν πίνακα κατακερματισμού για το πρώτο επίπεδο των N-grams. Ο πίνακας αυτός αποτελούταν από NOOFBUCKETS (4) στοιχεία (buckets) και κάθε τέτοιο στοιχείο περιλάμβανε έναν πίνακα από NOOFCELLS (4) dataNodes (cells). Ο πίνακας κατακερματισμού υπήρχε μέσα σε μια δομή hashTable, ένας δείκτης στην οποία δημιουργούταν στη main. Στη δομή αυτή συμπεριλαμβανόταν και ένας ακέραιος bucketToBeSplit για τον αριθμό του bucket το οποίο θα έπρεπε να διαχωριστεί στα 2 (split) μετά από υπερχείληση. Το μέγεθος του πίνακα διπλασιαζόταν μετά από δημιουργία 2*length buckets. Στη συνάρτηση getBucketFromHash μετατρέπαμε τον κάθε χαρακτήρα μιας λέξης σε ακέραιο και στη συνέχεια τους προσθέταμε και διαιρούσαμε τον τελικό αριθμό με το πλήθος των buckets για να επιστραφεί το bucket που αντιστοιχεί στη λέξη.

Παρακάτω φαίνεται η διαμόρφωση του αρχικού πίνακα μετά την εισαγωγή των φράσεων:

- this is a cat
- this is a
- this is a dog
- the fox is bad
- hello

Οι χρωματισμένοι κόμβοι είναι final.



Εικόνα 8: hashing example

Bloom Filter

Για την αποφυγή των διπλότυπων N-grams σε ένα ερώτημα αναζήτησης, σταματήσαμε να χρησιμοποιούμε πίνακα αποθήκευσης των εκτυπωμένων φράσεων και υλοποιήσαμε τη δομή Bloom Filter. Αυτή η δομή περιλαμβάνει τον αριθμό των συναρτήσεων κατακερματισμού (5) και έναν πίνακα μεγέθους HASH_SIZE (αρχικά 10000) τύπου bool.

Ως hashing στο bloomfilter χρησιμοποιήσαμε την MurmurHash3_x64_128. Η συγκεκριμένη διαιρεί το κλειδί σε chunks και μέσω rotation επιστρέφει ένα hash 128bit, ιδανικό για 64-bit μηχάνημα.

Top-k N-grams

Μας ζητήθηκε να εκτυπώνονται, μετά το τέλος της κάθε ριπής ερωτήσεων τα k N-grams που εκτυπώθηκαν περισσότερο. Γι' αυτό δημιουργήσαμε μια δομή topKArray που αποτελείται από έναν πίνακα από topKStruct. Κάθε κόμβος του πίνακα περιλαμβάνει μια συμβολοσειρά για το N-gram και έναν ακέραιο για τις εμφανίσεις αυτού του N-gram. Προκειμένου να μην εισάγονται διπλότυπα N-grams από ένα Q, χρησιμοποιήσαμε πάλι μια δομή Bloom Filter. Κάθε φορά επίσης που εισαγάγαμε ένα στοιχείο στον πίνακα χρησιμοποιούσαμε heapsort (με βάση τη συμβολοσειρά). Στο τέλος της ριπής, ταξινομούσαμε τον πίνακα με βάση τον ακέραιο (με heapsort) και εκτυπώναμε τα k πρώτα στοιχεία. Σε περίπτωση που το k ξεπερνούσε τον αριθμό των στοιχείων του πίνακα, εκτυπώναμε όλα τα στοιχεία.

Στατικό trie

Για τη δημιουργία του στατικού trie, υλοποιήσαμε κάποιες συναρτήσεις που έλεγχαν αν θα έπρεπε να συμπιεστούν δύο ή περισσότεροι κόμβοι και αν ήταν εφικτό, τους συμπίεζαν. Έτσι δημιουργήσαμε και μια ακόμα συνάρτηση για αναζήτηση η οποία καλούταν μόνο όταν το trie ήταν στατικό. Αυτό το καταλαβαίναμε από τα init_files των οποίων η πρώτη γραμμή ήταν είτε "DYNAMIC" είτε "STATIC".

Αρχεία δομών και συναρτήσεων

Τα καινούρια αρχεία που υλοποιήθηκαν σε αυτό το μέρος ήταν:

- bloomfilter.c : αρχικοποίηση filter (initializeFilter), επιστροφή hashes από Murmur (getHashesMurmur), υπολογισμός κ-οστής hash (kthHash), εισαγωγή στο filter (addFilter), έλεγχος ύπαρξης ngram (possiblyContains), διαγραφή filter (freeFilter).
- hashTable.c : δημιουργία γραμμικού hash (createLinearHash), καταστροφή γραμμικού hash (destroyLinearHash), εισαγωγή στο γραμμικό hash (insertTrieNode_insertTrieNodeAgain), αναζήτηση κόμβου σε bucket (lookupTrieNode), αρχικοποίηση bucket (initializeBucket), επιστροφή σωστού bucket από hashing (getBucketFromHash) , αναδιανομή κόμβων σε buckets (splitBucket), αύξηση level (levelUp), διαγραφή ενός bucket και όλων των nextWordArray του (deletionSortBucket), εκτύπωση ενός bucket (printBucket), εκτύπωση όλου του hashTable (printBuckets), αύξηση αριθμού κελιών σε ένα bucket (createOverflowCells), επιστροφή συγκεκριμένου κελιού (getCell), διαγραφή ενός bucket χωρίς την διαγραφή όλων των nextWordArray του (deletionSortWithoutErasingNextArray).
- topK.c : αρχικοποίηση topKArray (initializeTopKArray), διπλασιασμός topKArray (doubleTopKArray), αρχικοποίηση topKStruct (initializeTopKStruct), εισαγωγή στο topKArray (insertTopArray), εκτύπωση δοσμένων topK από το topKArray (printTopK), εκτύπωση όλων των topK από το topKArray (printFullArrayTop), καταστροφή topKArray (destroyTopArray), ταξινόμηση topKArray με HeapSort (HeapSort_BuildHeap), ταξινόμηση ως προς τα strings

(HeapifyStrings), ταξινόμηση ως προς τα occurences (HeapifyIntegers), αύξηση του occurences ενός topKStruct αν το string υπάρχει (binarySearchTopK).

- compress.c : ανακατασκευή δομής συμπιέζοντας τους κόμβους (recreateStructure), συμπίεση δύο κόμβων (compress), αναδρομική ανακατασκευή δομής σε κάθε arrayOfStruct (recursiveCompression), έλεγχος συμπίεσης και ύστερα συμπίεση (checkForCompression), επιστροφή ενός array of strings ανάλογα με τα πόσα αποτελείται ένας κόμβος (getNgramFromNode).

Προστέθηκαν επιπλέον οι συναρτήσεις:

- struct.c : αρχικοποίηση dataNode (initializeDataNode), εκτύπωση πίνακα staticArray (printStaticArray), διαγραφή επιπέδου μίας ολόκληρης φράσης (deleteArray1Layer).
- func.c: αναζήτηση ngram μετά από compress (search_ngram_StaticVersion).
-

Unit Testing

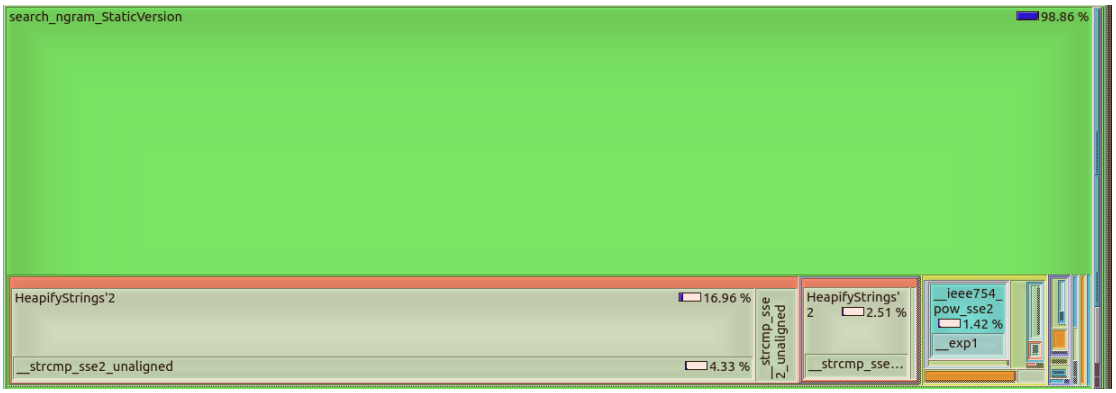
Όσων αφορά το Unit Testing, συνεχίσαμε να χρησιμοποιούμε το framework CuTest. Πρέπει να αναφερθεί ότι δεν υλοποιήσαμε όλες τις συναρτήσεις ξεχωριστά, καθώς ορισμένες καλούνταν η μία από την άλλη.

Στατιστικά εκτέλεσης

ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ	**ΧΡΟΝΟΙ ΕΚΤΕΛΕΣΗΣ**	**ΟΡΘΟΤΗΤΑ ΑΠΟΤΕΛΕΣΜΑΤΩΝ (✓)**
small_static	44.18 sec	✓
small_dynamic	54.35 sec	✓
medium_static	15 min	X
medium_dynamic	17 min	X

Χρονικά διαγράμματα

Περαιτέρω πληροφορίες σχετικά με τη χρονική διάρκεια των συναρτήσεων φαίνονται στα παρακάτω διαγράμματα.

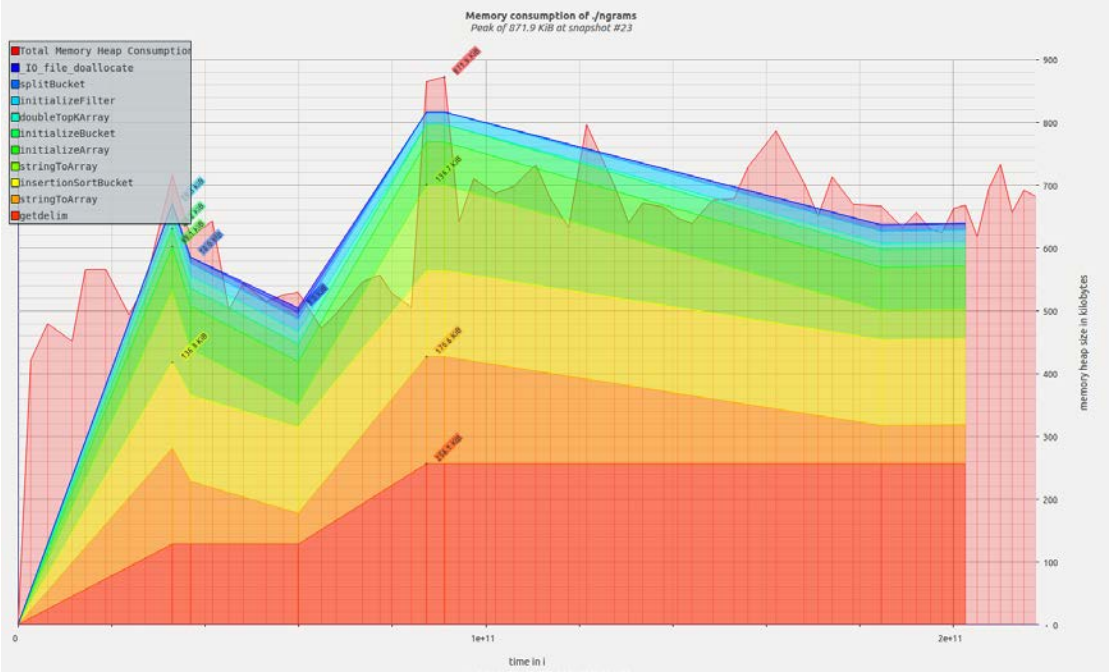


Εικόνα 9: small_static

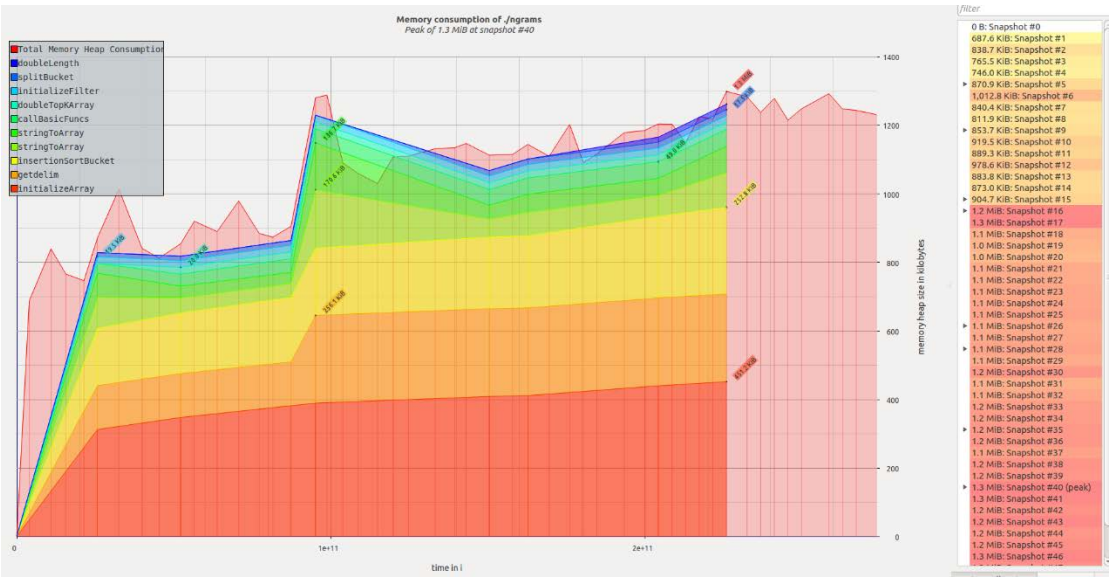
Εικόνα 10: small_dynamic

Χωρικά διαγράμματα

Τα διαγράμματα χώρου μνήμης φαίνονται παρακάτω.



Εικόνα 11: small_static



Εικόνα 12: small_dynamic

Σχόλια Εξεταστή

Τα μόνα αρχεία που έβγαζαν σωστά αποτελέσματα ήταν τα small_static και small_dynamic. Ο εξεταστής μας πρότεινε να αυξήσουμε το μέγεθος του πίνακα στη δομή Bloom Filter και/ή τον αριθμό των συναρτήσεων κατακερματισμού.

Διορθώσεις

Με τη χρήση της εντολής callgrind μπορέσαμε να βρούμε διάφορα σημεία τα οποία έκαναν τον κώδικά μας πιο αργό και να τα αλλάξουμε. Τα πιο σημαντικά από αυτά είναι ότι στη συνάρτηση getBucketFromHash μετατρέψαμε απευθείας τη συμβολοσειρά σε ακέραιο, αντί να προσθέτουμε ένα-ένα τους χαρακτήρες της. Επίσης στις συναρτήσεις search και search_static αφαιρέσαμε τη δημιουργία ενός καινούριου πίνακα από λέξεις για το query.

Η αλλαγή του μεγέθους του πίνακα στο Bloom Filter από 10000 σε 50000 σε συνδυασμό με την αλλαγή ορισμένων συνθηκών στις search, οδήγησε στην σωστή εκτέλεση του αρχείου medium_static. Στη συνέχεια ξαναστείλαμε στον υπεύθυνο καθηγητή την εργασία για να ελέγξει την πρόοδό μας.

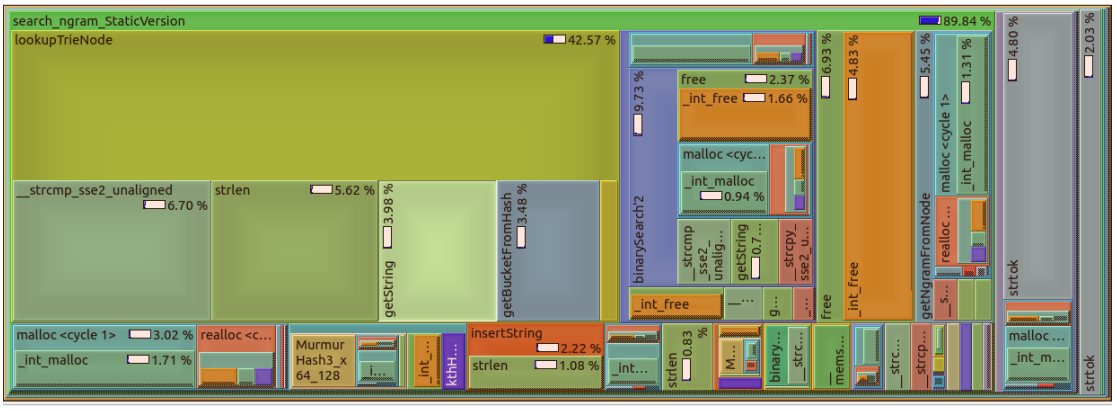
Καταφέραμε να μειώσουμε, όμως, ακόμα περισσότερο το χρόνο εκτέλεσης μειώνοντας τις δυναμικές δεσμεύσεις μνήμης και χρησιμοποιώντας στις συναρτήσεις search insertionSort αντί για heapsort. Αλλάξαμε επίσης το μέγεθος του πίνακα στο Bloom Filter σε 70000.

Με λίγες ακόμα αλλαγές και προσεκτικό έλεγχο καταφέραμε να εκτελέσουμε σωστά όλα τα δοσμένα αρχεία προς είσοδο (small_static, small_dynamic, medium_static, medium_dynamic, large_static, large_dynamic).

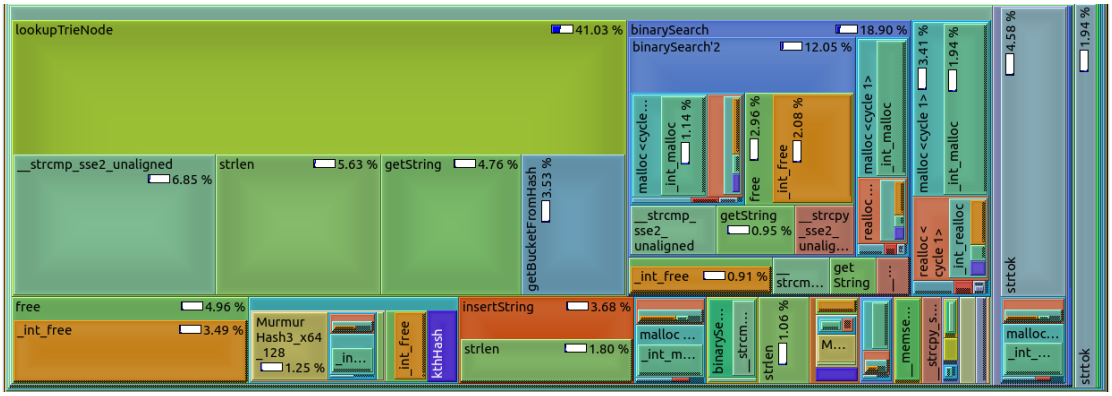
Στατιστικά εκτέλεσης

ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ	**ΧΡΟΝΟΙ ΕΚΤΕΛΕΣΗΣ**	**ΟΡΘΟΤΗΤΑ ΑΠΟΤΕΛΕΣΜΑΤΩΝ (✓)**
small_static	1.5 sec	✓
small_dynamic	1.9 sec	✓
medium_static	55 sec	✓
medium_dynamic	58 sec	✓
large_static	1.30 min	✓
large _dynamic	1.40 min	✓

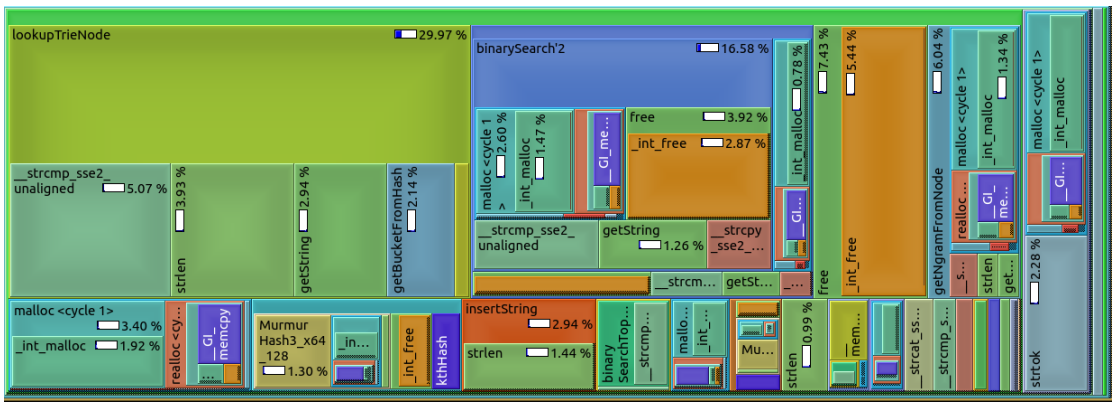
Παρακάτω παρουσιάζονται τα αντίστοιχα διαγράμματα χρόνου.



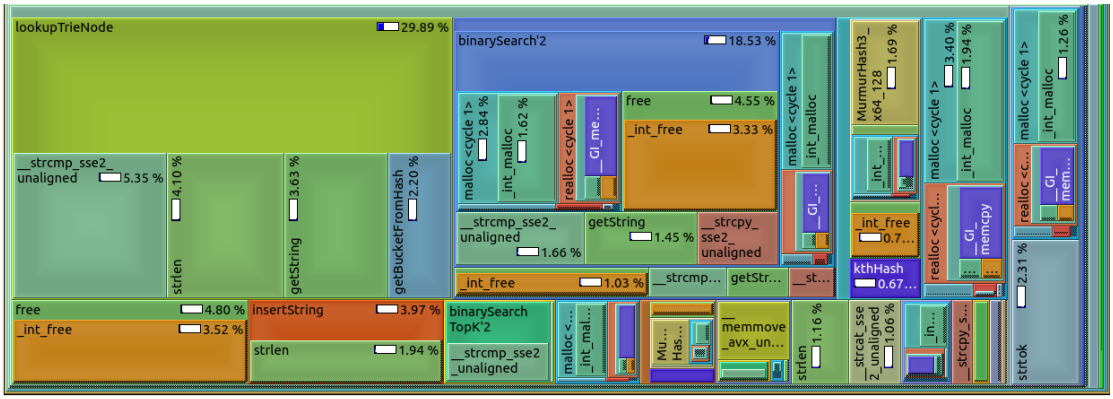
Εικόνα 13: small_static



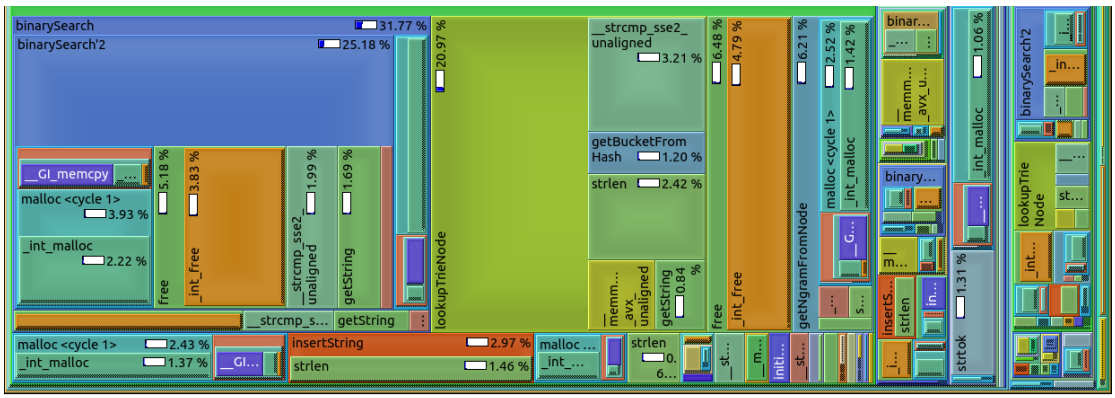
Εικόνα 14: small_dynamic



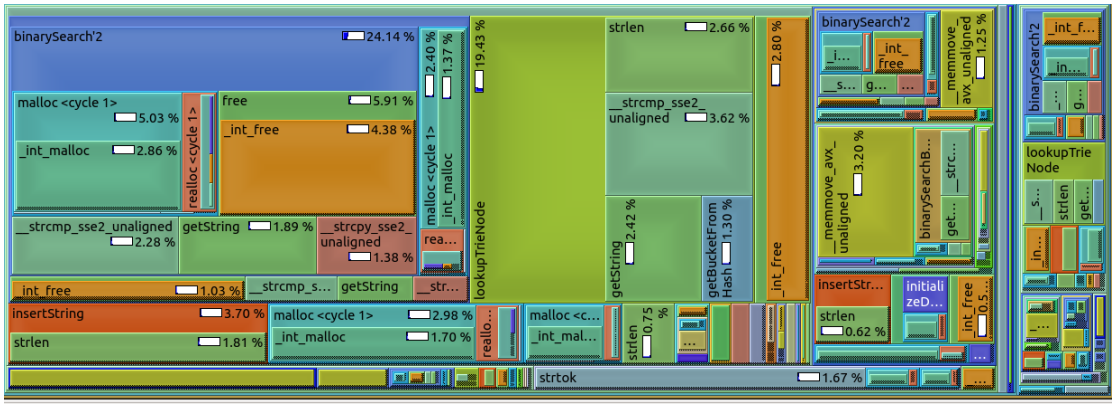
Εικόνα 15: medium_static



Εικόνα 16: medium_dynamic



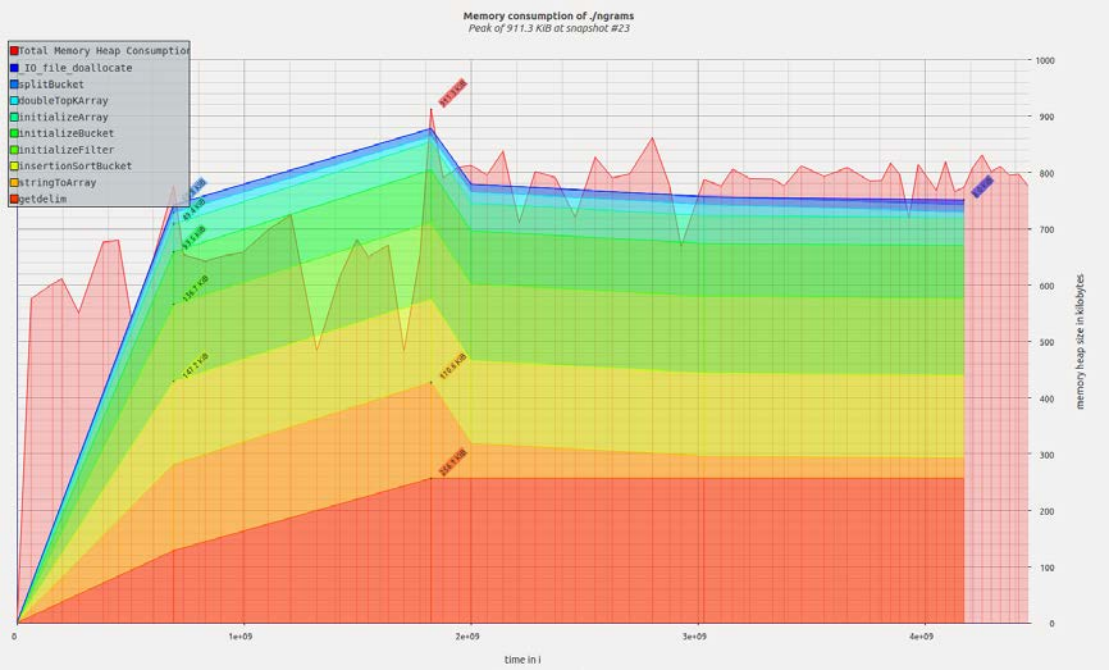
Εικόνα 17: large_static



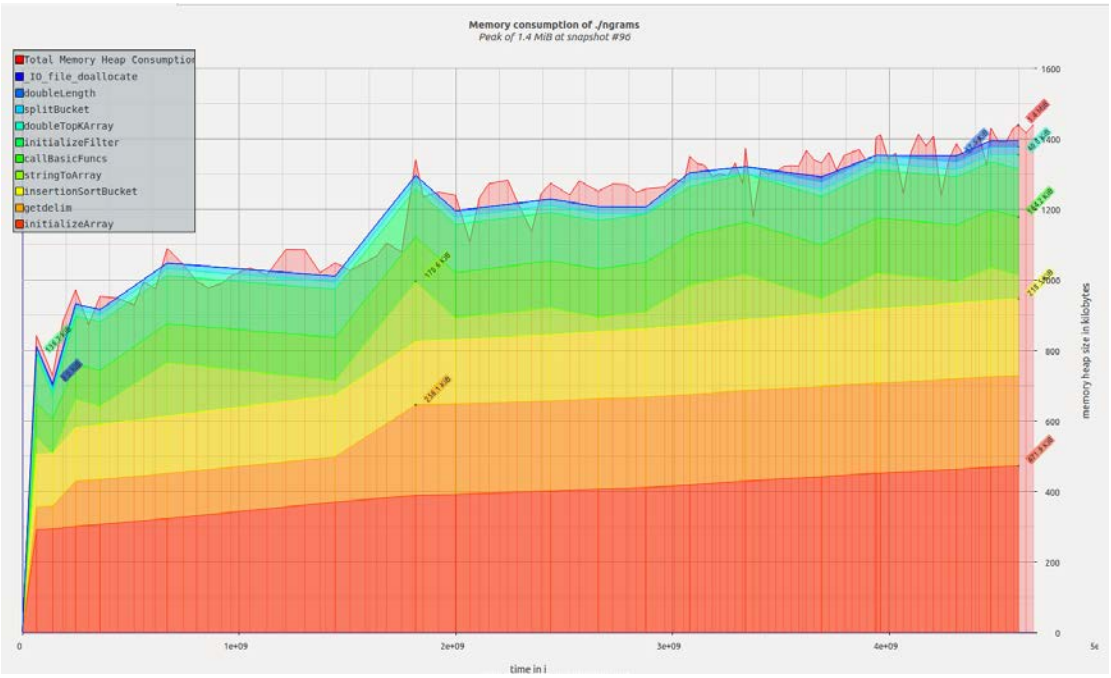
Εικόνα 18: large_dynamic

Χωρικά διαγράμματα

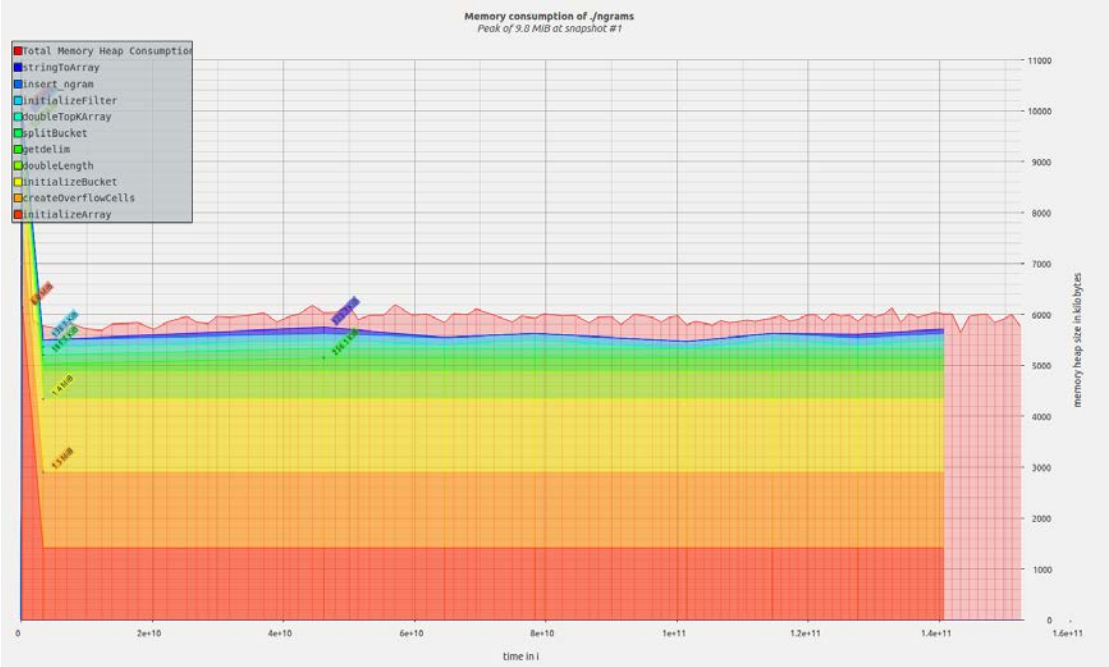
Τα αντίστοιχα διαγράμματα χώρου είναι:



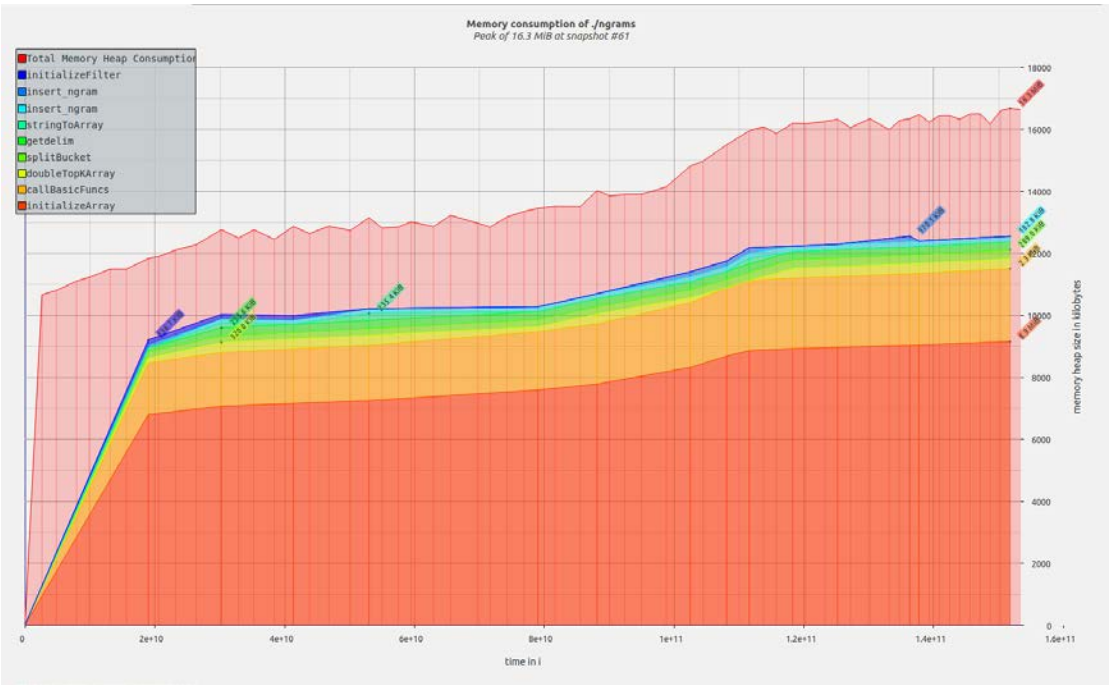
Εικόνα 19: small_static



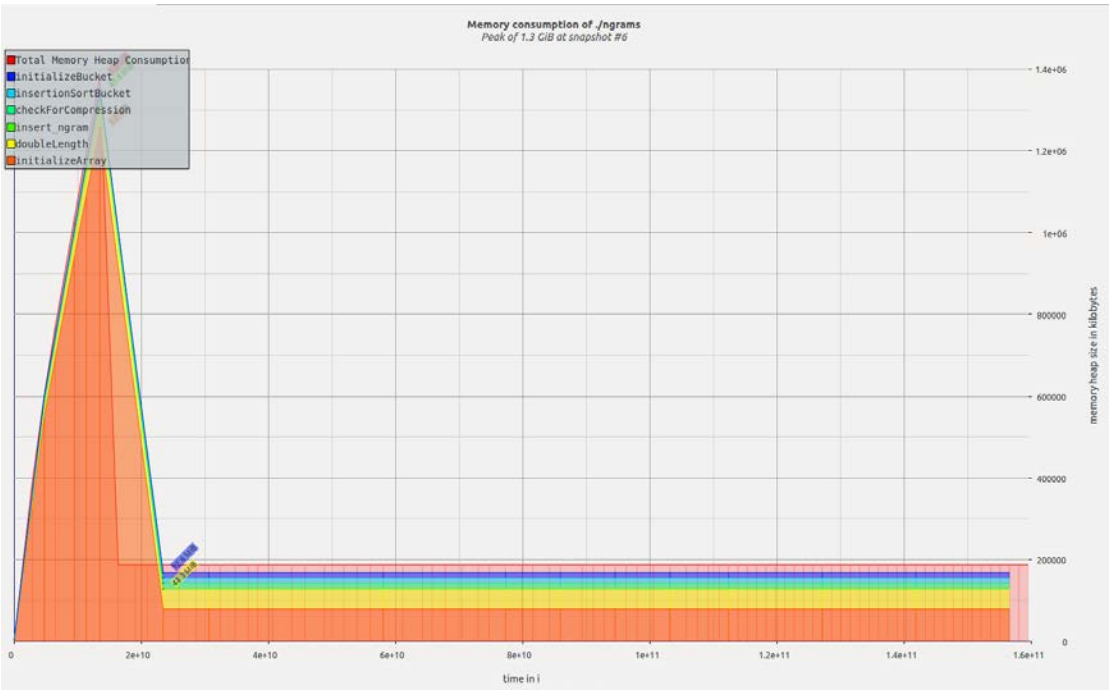
Εικόνα 20: small_dynamic



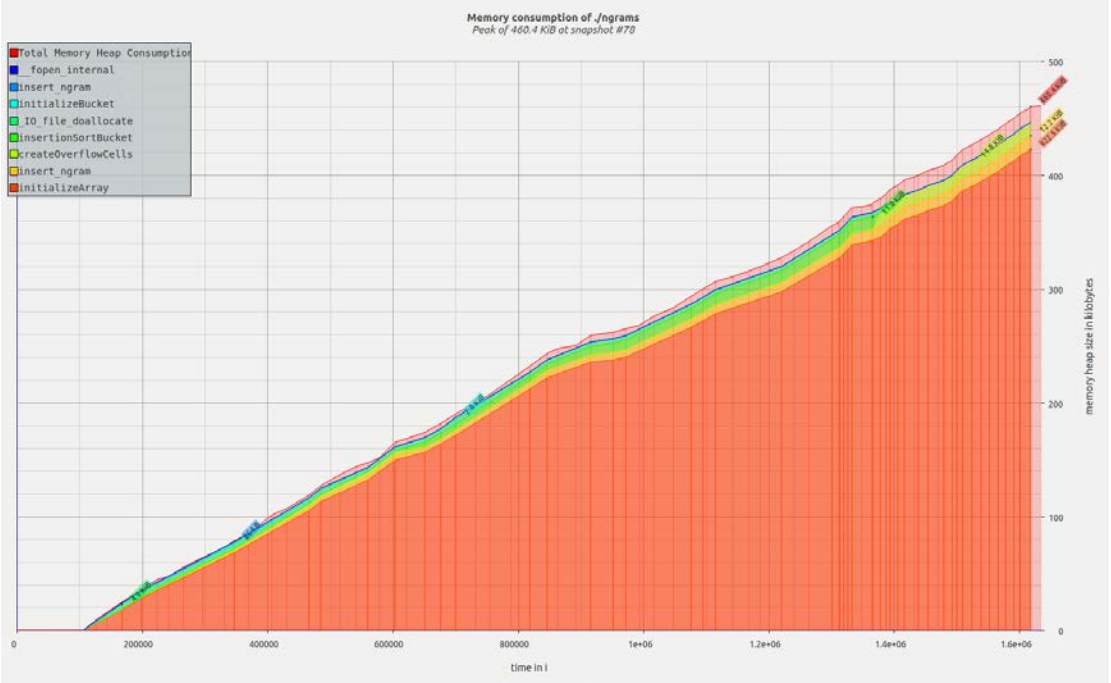
Εικόνα 21: medium_static



Εικόνα 22: medium_dynamic



Εικόνα 23: large_static



Εικόνα 24: large_dynamic

N-grams detection: Part 3

Δομές

JobScheduler

Στο τρίτο μέρος της εργασίας, έπρεπε να δημιουργήσουμε μια δομή JobScheduler, με τη βοήθεια της οποίας, τα ερωτήματα Q θα λύνονται παράλληλα από νήματα (threads). Η δομή JobScheduler αποτελείται από 3 structs: Την δομή του Job το οποίο περιέχει την συνάρτηση που πρέπει να κληθεί, τα ορίσματα της συνάρτησης σε έναν πίνακα από void* (δηλαδή είναι τύπου void**) και έναν δείκτη προς τον επόμενο κόμβο Job. Ο δείκτης αυτός μας χρησιμεύει για την υλοποίηση της ουράς. Η ουρά (struct Queue) είναι μία συνδεδεμένη λίστα η οποία αποτελείται από 3 μέλη, το μέγεθος (size), έναν δείκτη στην κεφαλή της ουράς από όπου και γίνεται η εξαγωγή (head) και έναν δείκτη στον τελευταίο κόμβο που βοηθά στην εισαγωγή νέου κόμβου Job (tail). Τέλος, υπάρχει η δομή JobScheduler η οποία κρατά ως πληροφορία τον αριθμό των νημάτων (executionThreads), την ουρά FIFO (jobQueue), τα νήματα (tids), ένα condition variable που ουσιαστικά ενημερώνει τα νήματα για αλλαγή κατάστασης στην ουρά (notifyThreads), ένα mutex για το κλείδωμα της ουράς (lockQueue) και μία flag που μας ενημερώνει πως η εισαγωγή έχει ολοκληρωθεί.

Αλλαγή σειράς εκτέλεσης εντολών ‘A’, ‘D’, ‘Q’

Για να είναι πιο αποτελεσματική η παραλληλία των queries, για κάθε ριπή εντολών, εκτελέσαμε πρώτα τις εισαγωγές και διαγραφές στον πίνακα κατακερματισμού και στη συνέχεια όλες τις αναζητήσεις. Δημιουργήσαμε λοιπόν μια δομή arrayOfInstructions η οποία αποτελούταν από έναν πίνακα από κόμβους instruction. Κάθε τέτοιος κόμβος περιλάμβανε ένα χαρακτήρα για τον τύπο της εντολής, την ίδια την εντολή, έναν αύξων αριθμό και έναν αύξοντα ακέραιο ο οποίος χρησίμευε μόνο για τη σειρά των queries (αρχικοποίηση στο -1 για ερωτήματα ‘A’ και ‘D’). Για κάθε ριπή ερωτημάτων, αρχικοποιούσαμε τη δομή arrayOfInstructions και εισαγάγαμε σε αυτή τις εντολές. Όταν η τρέχουσα εντολή ήταν ‘F’ αλλάζαμε τη δομή του πίνακα με τη συνάρτηση ”rearrangeArray“, έτσι ώστε να εκτελεστούν αρχικά οι εντολές εισαγωγής και διαγραφής και έπειτα όλες οι εντολές εύρεσης.

Μετέπειτα, προστέθηκε στις δομές του batch_handler.h και οι αντίστοιχες δομές που έχουν σχέση με την παραλληλοποίηση του στατικού trie. Για κάθε ριπή ερωτημάτων('Q') αρχικοποιούσαμε τη δομή arrayOfInstrStatic και κάναμε εισαγωγή τις αντίστοιχες εντολές από κάθε getline. Ο κάθε κόμβος του συγκεκριμένου πίνακα είναι τύπου instructionStatic, ο οποίος και αυτός με την σειρά του περιέχει το ngram, τον τύπο εντολής και έναν αριθμό ο οποίος ξεκινά από το -1. Στο τέλος της ριπής και αν το αρχείο είναι STATIC εκτελούμε τον JobScheduler.

Εισαγωγή N-gram

Για την ορθή λειτουργία της εισαγωγής προσθέσαμε στη δομή dataNode δύο ακεραίους. Ο πρώτος, additionVersion, αποθήκευε τον αύξοντα αριθμό της εντολής (σε περίπτωση αρχικοποίησης (init) είναι 0), ενώ ο δεύτερος, finalSinceAdditionVersion, έπαιρνε τιμή διαφορετική του -1 μόνο για final κόμβους. Επομένως για τελικούς κόμβους "finalSinceAdditionVersion == additionVersion".

Διαγραφή N-gram

Όσων αφορά τη διαγραφή, πρόσθεσα στη δομή dataNode, επίσης δύο ακεραίους (deletionVersion, notFinalInDeletionVersion) οι οποίοι έπαιρναν ως τιμή τον αύξοντα αριθμό της εντολής διαγραφής, και αρχικοποιούνταν στο -1. Επιπλέον σταματήσαμε να διαγράφουμε τους κόμβους από το hashTable και θέταμε τιμές στις δύο αυτές μεταβλητές των κόμβων προς διαγραφή.

Αναζήτηση N-gram

Εφόσον οι αναζητήσεις θα εκτελούνταν μετά τις εντολές εισαγωγής και διαγραφής, πρόσθεσα ορισμένους ελέγχους, για να εξακριβώσουμε ότι θα εκτυπώνονται κόμβοι που έχουν εισαχθεί τη σωστή στιγμή και δεν έχουν διαγραφεί. Αυτοί οι έλεγχοι είναι:

- Element->isFinal (το στοιχείο είναι τελικός κόμβος)
- queryNum>=Element->additionVersion (η εντολή αναζήτησης έπεται της εισαγωγή του κόμβου)
- queryNum>=Element->finalSinceAdditionVersion (η εντολή αναζήτησης πραγματοποιείται μετά από τον ορισμό του κόμβου ως τελικό)
- Element->finalSinceAdditionVersion!=-1 (ο κόμβος έχει όντως οριστεί ως τελικός)
- queryNum<Element->deletionVersion || Element->deletionVersion==-1 (η εντολή αναζήτησης προηγείται την διαγραφή του κόμβου ή ο κόμβος δεν έχει διαγραφεί ποτέ)
- queryNum < Element->notFinalInDeletionVersion || Element-> notFinalInDeletionVersion==-1 (η εντολή αναζήτησης προηγείται τον ορισμό του κόμβου ως μη τελικό ή ο κόμβος δεν ποτέ γίνει not final)

Διαγραφή κόμβων

Μετά το τέλος της εκτέλεσης των εντολών, καταστρέφαμε τους κόμβους προς διαγραφή με τη συνάρτηση "restructHashTable", έτσι ώστε να μην υπάρχουν «άχρηστοι» κόμβοι για το επόμενο batch.

Αρχεία δομών και συναρτήσεων

Τα καινούρια αρχεία που υλοποιήσαμε σε αυτό το μέρος ήταν:

- batch_handler.c : αρχικοποίηση δομής (initializeInstructionArray), διπλασιασμός πίνακα (doubleInstructionArray), εισαγωγή στον πίνακα (insertInstructionArray), διαγραφή δομής (destroyInstructionArray), αρχικοποίηση εντολής (initializeInstruction), εκτύπωση εντολής (printInstruction), εκτύπωση πίνακα (printInstructionArray), αναδιάταξη πίνακα (rearrangeArray), μετακίνηση στοιχείων (moveUp) αρχικοποίηση δομής πίνακα (initializeInstrStaticArr), αρχικοποίηση δομής static (initializeInstrStatic), διπλασιασμός δομής static (doubleInstrStaticArray), εισαγωγή δομής static στον πίνακα (insertInstrStaticArray), διαγραφή static κόμβου (destroyInstructionStatic), διαγραφή static πίνακα (destroyInstrStaticArray).
- JobScheduler.c: αρχικοποίηση Job (initializeJob), διαγραφή Job (deleteJob), αρχικοποίηση ουράς Queue (initializeQueue), εισαγωγή στην ουρά (pushToQueue), εξαγωγή από την ουρά (popFromQueue), καταστροφή ουράς (destroyQueue), εκτύπωση ουράς (printQueue), αρχικοποίηση JobScheduler (initializeScheduler), εισαγωγή Job στον JobScheduler (submitJob), εκτέλεση όλων των Jobs (executeAllJobs), αναμονή μέχρι την εισαγωγή όλων των Jobs (waitAllTasksFinish), καταστροφή JobScheduler (destroyScheduler).

Προστέθηκαν επιπλέον οι συναρτήσεις:

- hashTable.c : αναδιάρθρωση πίνακα κατακερματισμού (restructHashTable) και αναδιάρθρωση πίνακα arrayOfStructs (restructArray).
- auxMethods.c: εκτέλεση εντολών για δυναμικό trie (executeDynamicArray) και έλεγχος αν ο κόμβος έχει παιδιά (hasChildren).
-

Unit Testing

Υλοποιήσαμε και τα υπόλοιπα αρχεία για έλεγχο συναρτήσεων στο φάκελο cutest-1.5. Κάποια unit tests που αφορούν τον JobScheduler δεν λειτουργούν όπως αναμενόταν και για αυτό είναι σχολιασμένος ο κώδικάς τους (CuTestJobScheduler/CuTestAuxMethods).

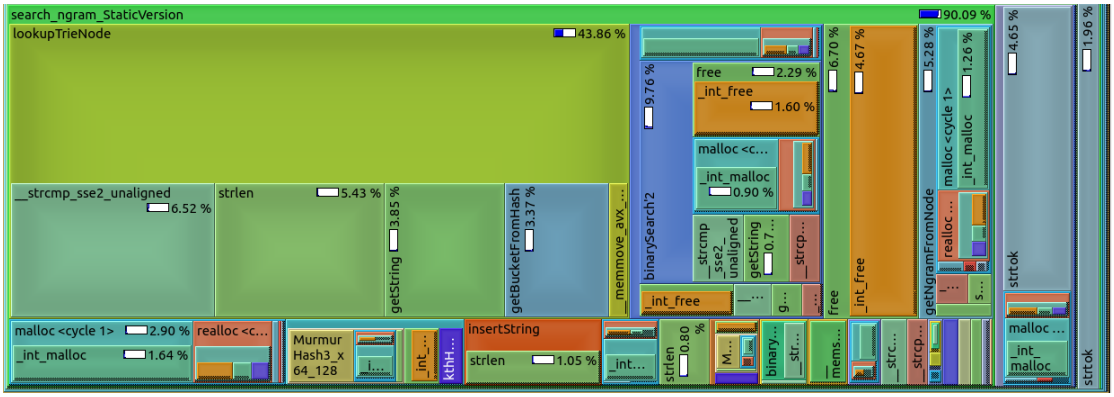
Πρώτη έκδοση τρίτου μέρους

Στατιστικά εκτέλεσης

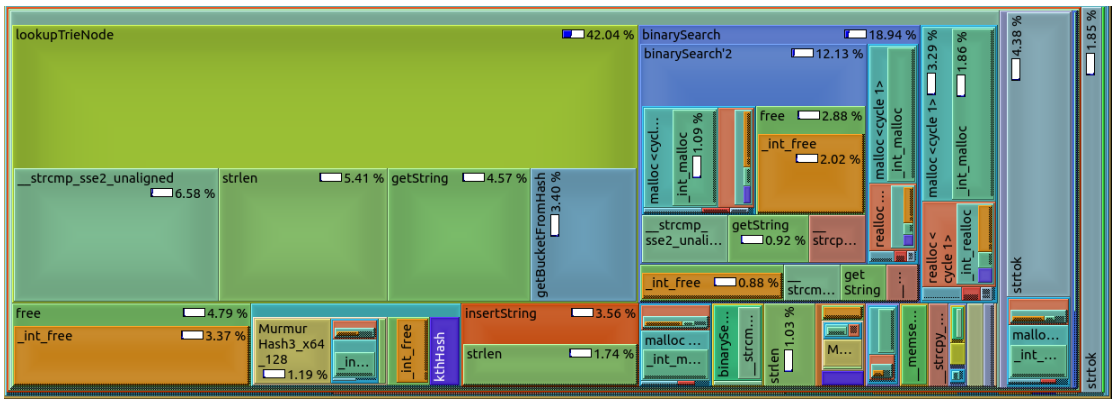
ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ	ΧΡΟΝΟΙ ΕΚΤΕΛΕΣΗΣ	ΟΡΘΟΤΗΤΑ ΑΠΟΤΕΛΕΣΜΑΤΩΝ (✓)
small_static	1.30 sec	✓
small_dynamic	1.20 sec	✓
medium_static	57 sec	✓
medium_dynamic	1 min	✓
large_static	1.30 min	✓
large _dynamic	1.40 min	✓

Χρονικά διαγράμματα

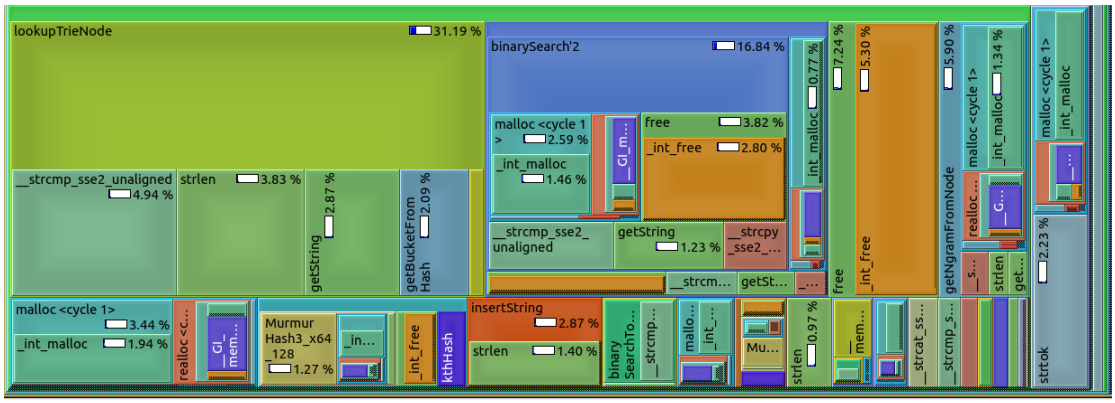
Παρακάτω παρουσιάζονται τα διαγράμματα χρόνου.



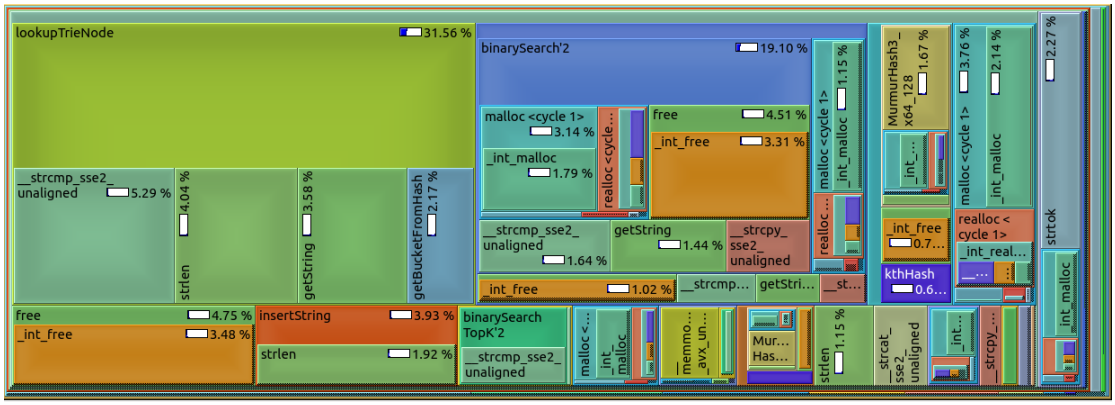
Εικόνα 25: small_static



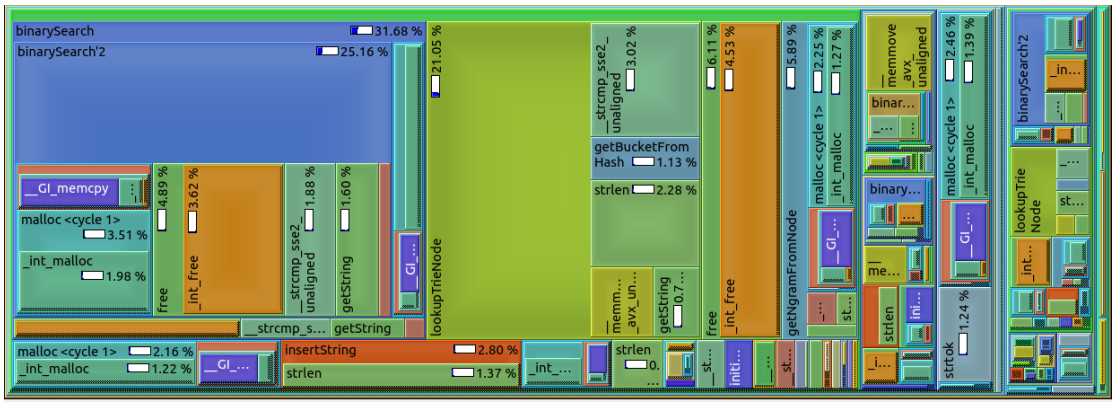
Εικόνα 26: small_dynamic



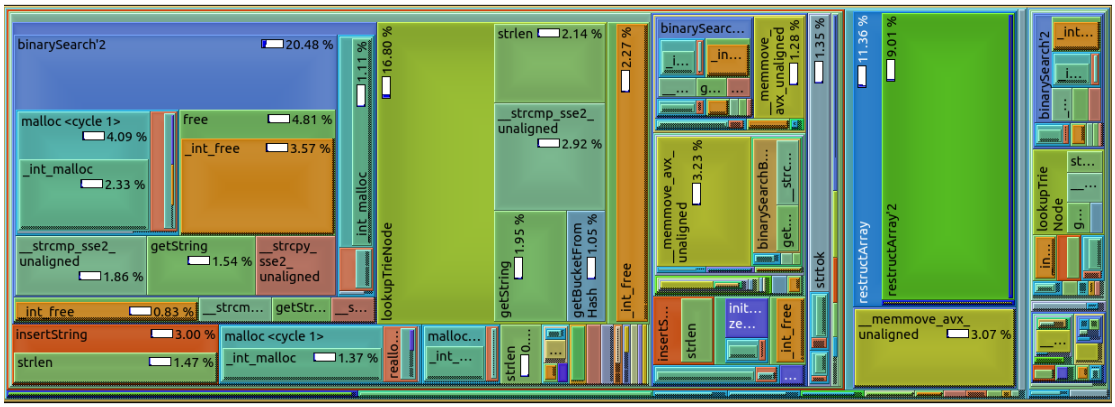
Εικόνα 27: medium_static



Εικόνα 28: medium_dynamic



Εικόνα 29: large_static



Εικόνα 30: large_dynamic

Δεύτερη έκδοση τρίτου μέρους

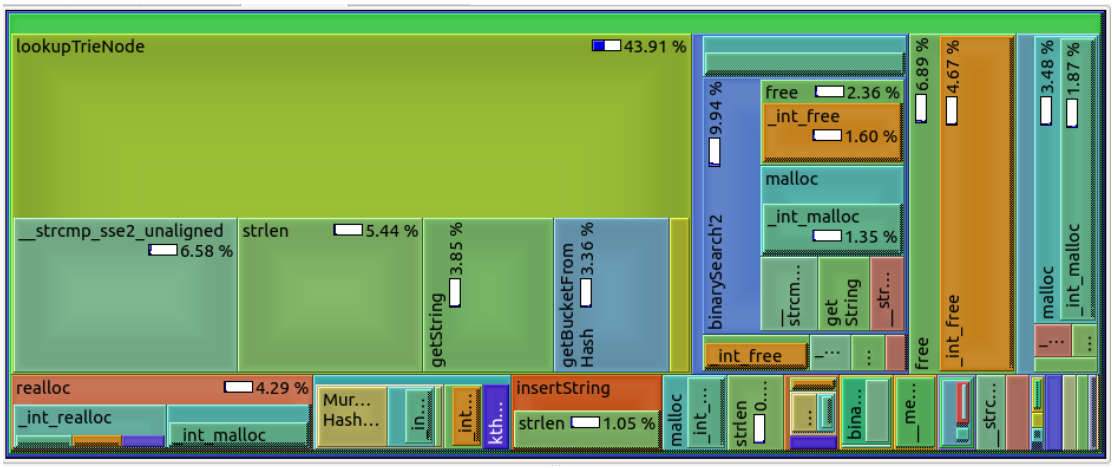
Στατιστικά εκτέλεσης

ΑΡΧΕΙΑ ΕΙΣΟΔΟΥ	ΧΡΟΝΟΙ ΕΚΤΕΛΕΣΗΣ	ΟΡΘΟΤΗΤΑ ΑΠΟΤΕΛΕΣΜΑΤΩΝ (✓)
small_static	1.31 sec	✓

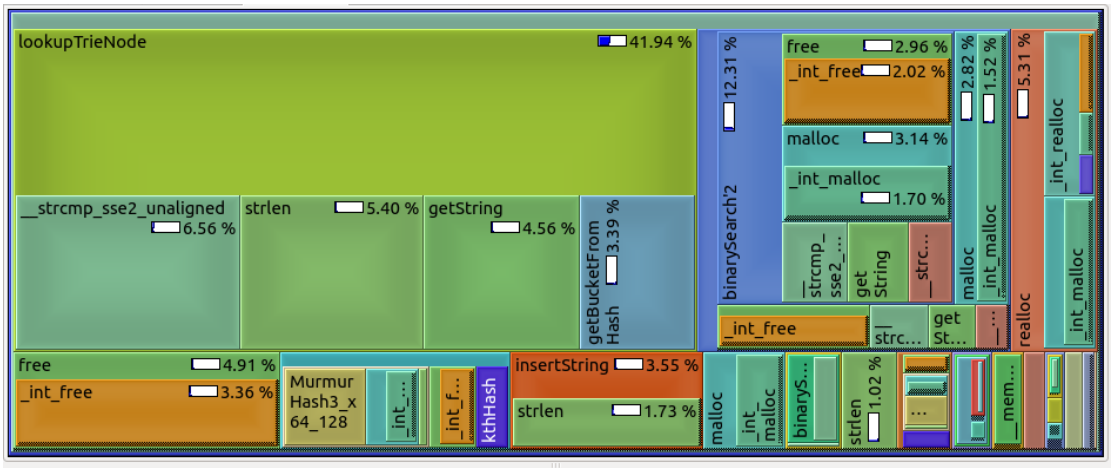
small_dynamic	1.29 sec	✓
medium_static	0.57 sec	✓
medium_dynamic	-	X
large_static	2.48 min	✓
large _dynamic	-	X

Χρονικά διαγράμματα

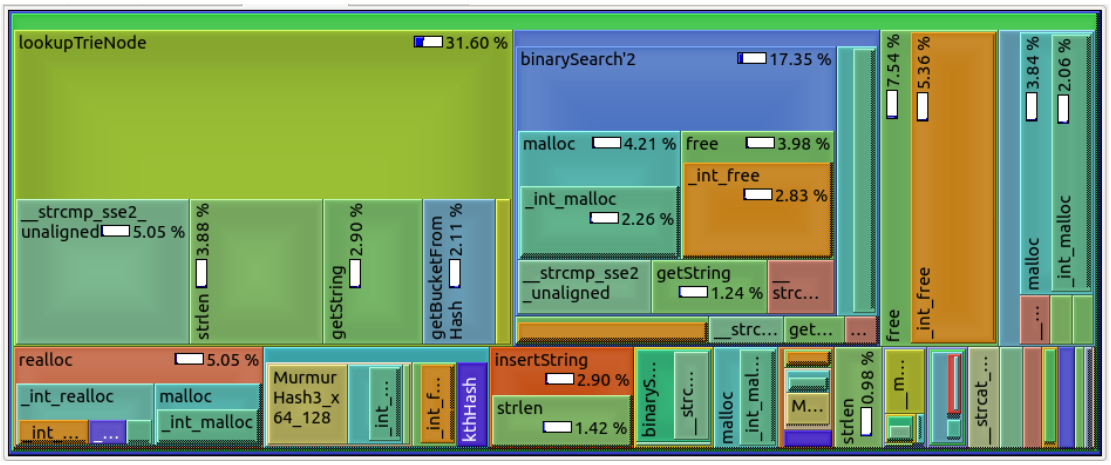
Τα χρονικά διαγράμματα για την δεύτερη έκδοση είναι:



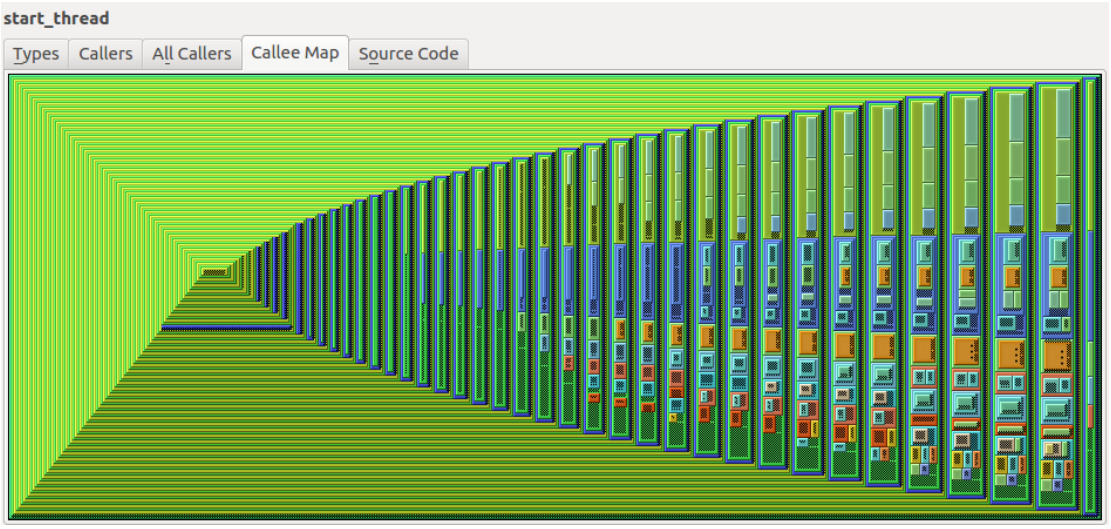
Εικόνα 31: small_static



Εικόνα 32: small_dynamic



Εικόνα 33: medium_static

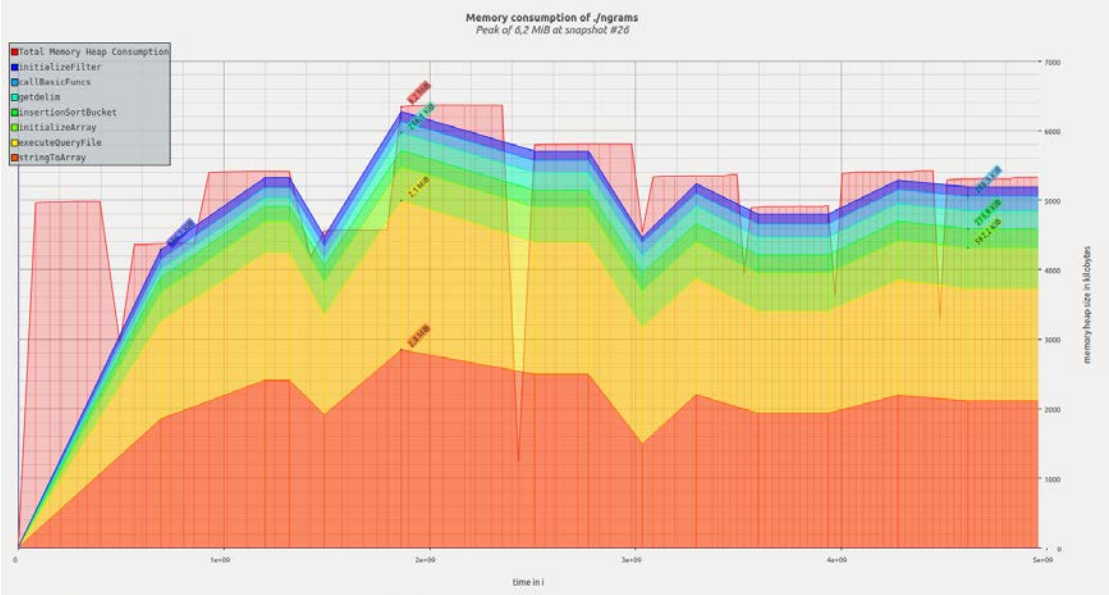


Εικόνα 34: medium_static thread display

Χωρικά διαγράμματα

Τα χωρικά διαγράμματα είναι:

Εικόνα 35: small_static



Εικόνα 36: small_dynamic

Επίλογος/Σχόλια

Για τη δημιουργία των διαγραμμάτων χρησιμοποιήσαμε τις εντολές callgrind και massif, ενώ για τη χρονομέτρηση των προγραμμάτων, την εντολή time. Το τρίτο μέρος της εργασίας δεν είναι πλήρως ολοκληρωμένο λόγω μειωμένου χρόνου. Παρόλα αυτά, προσπαθήσαμε όσο το δυνατόν να υλοποιήσουμε όσα περισσότερα μέλη της εργασίας μπορούσαμε. Για τον λόγο αυτό, χωρίσαμε την 3η εργασία σε δύο εκδόσεις, πριν την εισαγωγή των threads και την ολοκλήρωση της αλλαγής της σειράς των εντολών, και στην έκδοση που τα νήματα

δουλεύουν μόνο αν αρχικοποιήσουμε τον JobScheduler με ένα νήμα. Η δεύτερη έκδοση δουλεύει για τα αρχεία small_static, small_dynamic και medium_static, ενώ η πρώτη για όλα τα αρχεία. Γι' αυτό το λόγο παραδίδουμε και ένα συμπιεσμένο αρχείο "project_EarlierVersion.zip" που περιλαμβάνει το πρόγραμμα στην πρώτη έκδοση, δηλαδή πριν την εισαγωγή του Scheduler.

Για να εκτελέσουμε το πρόγραμμά μας, χρησιμοποιήσαμε "VMware Workstation Pro". Τα συστήματα που ελέγξαμε το πρόγραμμα ήταν: "Ubuntu 16.04 64bit", με μνήμη RAM 6 Gb και επεξεργαστές i5 και i7 intel.