Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα 2017-2018
Μέρος 3ο

Αργυρός Ιωάννης - 1115201200009
Γιαννούδης Αστέριος - 1115201200025
Κολυβάς Κωνσταντίνος - 1115201200066

Περιεχόμενα

Γλώσσα: C++11

Μεταγλώττιση:

```
$ cd build
$ cmake ..
$ make ngrams
```

Εκτέλεση:

```
$ ./ngrams
        -i <init_file>
        -q <query_file>
        [-t|--threads <int>
         -p|--parallel
         -c|--clean_up
         -h|--help]
```

Μεταγλώττιση και εκτέλεση των Unit Tests

```
$ make basic_tests
$ ./basic_tests
```

Καθάρισμα φακέλου build (-g για να μη σβηστούν τα αρχεία του googletest) :

```
$ sh clean.sh [-g]
```

1. Thread Pool - Job Scheduler

Γενικευμένη δομή εκτέλεσης εργασιών. Χωρίζεται στις παρακάτω Κλάσεις:

1.1 **`thread`**:

Abstract κλάση από την οποία κληρονομεί οποιαδήποτε κλάση θέλει να τρέξει σε νέο νήμα.

Αξιοσημείωτες συναρτήσεις:

`void thread::run()` :

Στην κλάση thread, αυτή η συνάρτηση είναι δηλωμένη ως pure virtual, δηλαδή κάθε κλάση που κληρονομεί από την thread, πρέπει να την κάνει override. Σε αυτή τη συνάρτηση, γράφεται ο κώδικας που θα τρέξει μέσα στο νέο νήμα.

`void thread::start()` :

Εκτελεί την συνάρτηση `thread::run()` σε νέο νήμα.

`void thread::join()` :

Περιμένει την φυσιολογική ολοκλήρωση εκτέλεσης της συνάρτησης run και καταστρέφει το νήμα.

1.2 **`task`**:

Abstract κλάση από την οποία κληρονομεί οποιαδήποτε κλάση θέλει να τρέξει μέσα στον Job Scheduler. Χρησιμοποιείται για πολυμορφισμό.

Αξιοσημείωτες συναρτήσεις:

`void task::run()` :

Στην κλάση task, αυτή η συνάρτηση είναι δηλωμένη ως pure virtual, δηλαδή κάθε κλάση που κληρονομεί από την task

πρέπει να την κάνει override. Σε αυτή τη συνάρτηση, γράφεται ο κώδικας που θα εκτελεστεί από τον Job Scheduler.

1.3 `work_queue`:

Η ουρά εργασιών του thread pool. Διαθέτει μία δομή ουράς (`mstd::queue<task *>`) η οποία περιέχει δείκτες σε αντικείμενα **`task`**, ένα mutex και ένα conditional variable.

Αξιοσημείωτες συναρτήσεις:

`void` `work_queue::add_task(`**`task`**` *)` :

Εισάγει ένα νέο **`task`** στην ουρά εργασιών.

`task` `*work_queue::next_task()` :

Επιστρέφει την επόμενη εργασία από την ουρά. Αν η ουρά είναι άδεια, τότε διακόπτει την εκτέλεση του νήματος που την κάλεσε μέχρι κάποια εργασία να εισαχθεί στην ουρά.

Λειτουργία:

Εισαγωγή : Αφού κλειδωθεί το mutex της ουράς εργασιών και εισαχθεί το στοιχείο, στέλνεται σήμα εισαγωγής, και αν κάποιο thread περιμένει διότι η ουρά ήταν άδεια, ξυπνάει και συνεχίζει την εκτέλεσή του.

Εξαγωγή : Αφού κλειδωθεί το mutex της ουράς εργασιών, ελέγχεται αν η ουρά είναι άδεια.
Αν είναι, τότε το νήμα σταματάει την εκτέλεσή του μέχρι να δεχθεί σήμα πως κάποιο στοιχείο εισήλθε στην ουρά.
Σε κάθε περίπτωση, το επόμενο στοιχείο της ουράς εξάγεται από εκείνη και επιστρέφεται.

1.4 `worker`:

Κληρονομεί από την κλάση `thread`. Δημιουργείται κατά την δημιουργία ενός thread pool και καταστρέφεται μόνο υπό εντολή του thread pool που το δημιούργησε.

Λειτουργία:

Εξάγει στοιχεία από την ουρά εργασιών και τα εκτελεί.
Αν η ουρά είναι άδεια, τότε η εκτέλεση διακόπτεται μέχρι να εισαχθεί κάποιο στοιχείο.
Μετά την εκτέλεση της κάθε εργασίας, αυξάνει μια κοινή μεταβλητή (`_num_finished`) κατά ένα (Απαραίτητο για την σωστή λειτουργία της συνάρτησης `thread_pool::wait_all()`).
Αν το στοιχείο που εξήχθη από την ουρά, είναι null, τότε σταματάει τελείως την εκτέλεσή του. Με αυτό τον τρόπο, το thread pool καταστρέφει κάθε νήμα - worker.

1.5 `thread_pool`:

Αυτή η κλάση είναι υπεύθυνη για τη διαχείριση της λειτουργίας του Job Scheduler. Δημιουργεί τα νήματα - workers, εισάγει στοιχεία στην ουρά εργασιών και καταστρέφει τα νήματα όταν χρειάζεται.

Κατά την δημιουργία του, δημιουργεί έναν πίνακα (`mstd::vector<worker *>`) N στοιχείων και εκκινεί τα νήματα - workers.

Αξιοσημείωτες συναρτήσεις:

`void thread_pool::add_task(task *t)` :

Εισάγει το στοιχείο στην ουρά εργασιών και αυξάνει κατά ένα μια εσωτερική μεταβλητή (`_num_assigned`) η οποία κρατάει πόσες δουλειές έχουν ανατεθεί μέχρι στιγμής.

`void thread_pool::finish()` :

Εισάγει N null εργασίες στην ουρά εργασιών. Κάθε ένα νήμα - worker, μόλις εξάγει μια null εργασία από την ουρά, διακόπτει την εκτέλεσή του, οπότε το thread_pool κάνει join όλα τα νήματα - workers και τα καταστρέφει.

`void thread_pool::wait_all()` :

Αρχικά ελέγχεται αν ο αριθμός των εργασιών που έχουν ανατεθεί είναι ίσος με τον αριθμό των εργασιών που έχουν τελειώσει. Αν είναι, τότε το πρόγραμμα επιστρέφει στην κανονική ροή εκτέλεσης. Διαφορετικά, η εκτέλεση του νήματος διακόπτεται και συνεχίζεται μόνο όταν κάποια εργασία ολοκληρωθεί. Τότε ελέγχεται ξανά η από πάνω συνθήκη. Αυτό επαναλαμβάνεται όσες φορές χρειαστεί, οπότε και η ροή του νήματος επανέρχεται στο κανονικό.

2. Σχεδιαστικές επιλογές

2.1. Clean up:

Εκτελέσαμε το πρόγραμμα κάνοντας clean up στο δέντρο μετά από κάθε batch και χωρίς να το κάνουμε. Οι διαφορές στο χρόνο εκτέλεσης (βλ. 3.1.1), ειδικά στο large dataset ήταν ιδιαίτερα μεγάλες (της τάξης του 50%) ενώ δεν αυξήθηκε αντίστοιχα η χρήση μνήμης (βλ. 4.2). Έτσι αποφασίσαμε να δίνουμε την επιλογή για clean up στον χρήστη (μέσω του flag -c).

2.1.1. Λειτουργία της clean up:

Κατά τη διάρκεια των διαγραφών, οι κόμβοι που θα διαγράφονταν, σημειώνονται για διαγραφή από την trie::delete_ngram. Κατά τη διάρκεια της trie::clean_up, αν βρεθεί κάποιος κόμβος ο οποίος έχει σημειωθεί για διαγραφή, τότε διαγράφεται (και όλα τα παιδιά του) από το δέντρο.

2.2. Βελτιστοποιήσεις:

Οι παρακάτω επιλογές/βελτιστοποιήσεις, μπορούν να ενεργοποιηθούν με την χρήση του ορίσματος -p κατά την εκτέλεση του προγράμματος. Αποφασίσαμε, μετά από δοκιμές, να συμπεριλάβουμε τις αλλαγές αυτές μετά από επιλογή του χρήστη, καθώς τα αποτελέσματα δεν ήταν πάντα εμφανώς καλύτερα με τη χρήση τους.

Παράλληλο Compress:

Για να επιτευχθεί η παράλληλη συμπίεση του αρχικού trie, χρησιμοποιούνται τα διαθέσιμα threads, όπου το κάθε ένα εκτελεί τον αλγόριθμο συμπίεσης για κάθε κόμβο παιδί της ρίζας. Με τον τρόπο αυτό, η συμπίεση γίνεται σε κάθε διαφορετικό υποδέντρο μετά τη ρίζα και έτσι δεν χρειάζονται mutexes για το συγχρονισμό των threads.

Παράλληλο Top K:

Η παράλληλη λειτουργία topk επιτυγχάνεται με τη χρήση πολλών hashtables, έναν για κάθε thread, στους οποίους μοιράζονται τα αποτελέσματα των queries. Κάθε thread θα αναλάβει να γεμίσει το δικό του hashtable με ένα μέρος από τις απαντήσεις που θα βρει στον πίνακα όλων των απαντήσεων. Στη συνέχεια, το master thread αναλαμβάνει να ενώσει όλους τους hashtables σειριακά. Από εκεί και έπειτα, η λειτουργία του topk συνεχίζεται ως είχε.

Παράλληλο Add Delete:

Για την εισαγωγή και διαγραφή Ngram - τα οποία δεν μπορούν να γίνουν παράλληλα - δημιουργείται ένα thread pool το οποίο περιέχει μόνο ένα νήμα - worker, στο οποίο νήμα εκτελούνται όλα τα add και delete σειριακά μεταξύ τους, αλλά παράλληλα με το υπόλοιπο πρόγραμμα (δηλαδή την ανάγνωση του αρχείου work).

3. Χρόνοι Εκτέλεσης

3.1. Διαφορές χρόνου:

Για το small dynamic και small static dataset, οι χρόνοι εκτέλεσης ήταν αρκετά μικροί με οποιαδήποτε αλλαγή για βελτιστοποίηση, οπότε δεν υπήρχε κάποια εμφανής διαφορά. Οπότε παρακάτω παρατίθενται οι διαφορές μόνο στα medium και large datasets.

3.1.1. Με το flag -O3 (σε second):

Medium:

Dynamic:

Part 1:	58,626410294
Part 2:	26,728223239
Part 3 με -p:	16,332112029
Part 3 χωρίς -p με -c:	16,424131173
Part 3 χωρίς -c με -p:	14,230002192

Static:

Part 1:	43,988567475
Part 2:	33,205213809
Part 3 με -p:	17,172595226
Part 3 χωρίς -p:	18,485156978

Large:

Dynamic:

Part 1:	Δεν τελείωσε
Part 2:	47,991847220
Part 3 με -p:	83,463725739
Part 3 χωρίς -p με -c:	89,143440400
Part 3 χωρίς -c με -p:	34,990499678

Static:

Part 1:	Δεν τελείωσε
Part 2:	49,361142799
Part 3 με -p:	26,848788636
Part 3 χωρίς -p:	29,479869347

3.1.2. Χωρίς το flag -O3 (σε second):

Medium:

Dynamic:

Part 1:	107,56642310
Part 2:	45,996001329
Part 3 με -p:	26,120819037
Part 3 χωρίς -p:	27,168173821
Part 3 χωρίς -c με -p:	23,429654795

Static:

Part 1:	87,790051885
Part 2:	56,966255359
Part 3 με -p:	28,382351415
Part 3 χωρίς -p:	28,053998795

Large:

Dynamic:

Part 1:	Δεν τελείωσε
Part 2:	78,267940449
Part 3 με -p:	110,073967368
Part 3 χωρίς -p:	117,455813840
Part 3 χωρίς -c με -p:	53,400778474

Static:

Part 1:	Δεν τελείωσε
Part 2:	72,192314300
Part 3 με -p:	40,839687829
Part 3 χωρίς -p:	43,753295033

Στα static datasets δεν έχουν νόημα οι μετρήσεις με -c, διότι δεν γίνονται προσθήκες/διαγραφές ngrams κατά την επεξεργασία του .work και έτσι δεν χρειάζεται να καθαριστεί το trie με την clean_up.

3.2 Παρατηρήσεις:

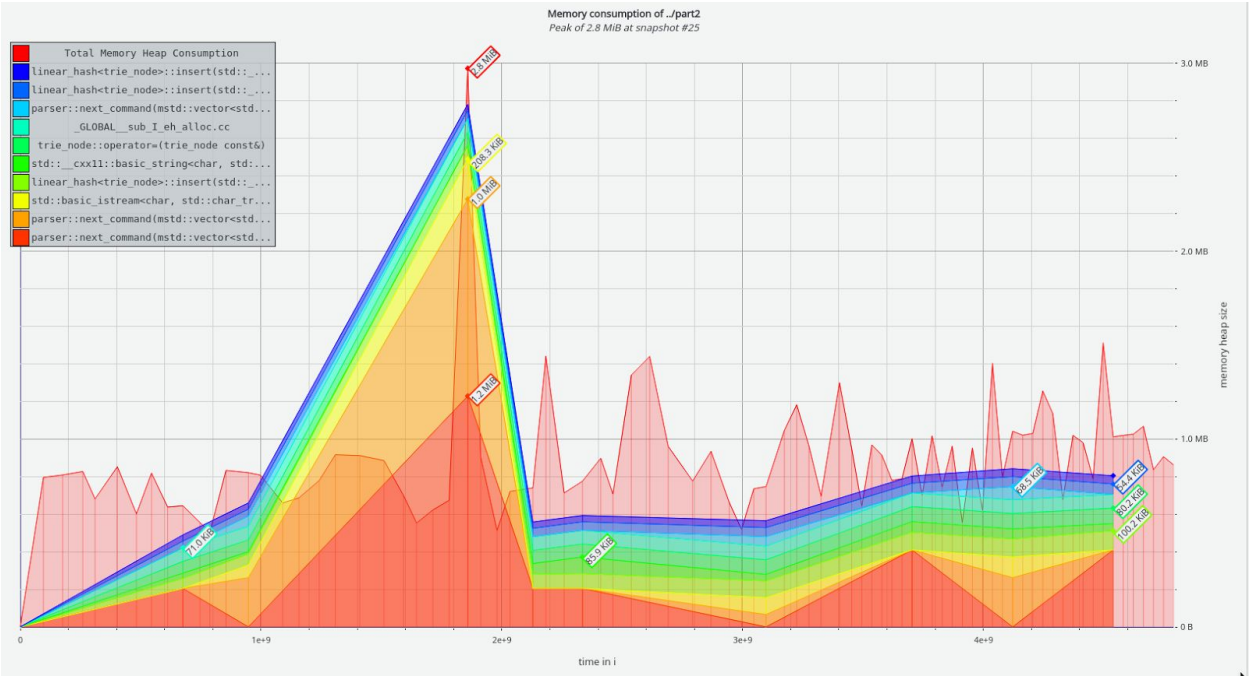
Παρατηρήσαμε πως στο medium dynamic - static και στο large static (με και χωρίς την χρήση του flag -O3), είδαμε συντριπτική διαφορά μεταξύ των δύο υλοποιήσεων, καθώς επωφεληθήκαμε από την παραλληλοποίηση των ερωτημάτων (Queries).

Αντίθετα, στο large dynamic, εξ'αιτίας των πολλών Addition/Deletion (998,690/199,947 αντίστοιχα) σε σχέση με τα Query (10,000), ο χρόνος εκτέλεσης του Part 3 είναι μεγαλύτερος σε σχέση με του Part 2, διότι η διαδικασία clean up του δέντρου μετά από κάθε batch (αν αυτή έχει ενεργοποιηθεί), προσθέτε1ι επιπλέον χρόνο, καθώς διασχίζεται όλο το δέντρο για να βρεθούν οι διαγεγραμμένοι κόμβοι. Πράγματι, μετρώντας μόνο το χρόνο εκτέλεσης των queries στο large dataset, στο part 2 χρειάστηκαν 15 secs για να ολοκληρωθούν, ενώ στο part 3 μόνο 3 secs.

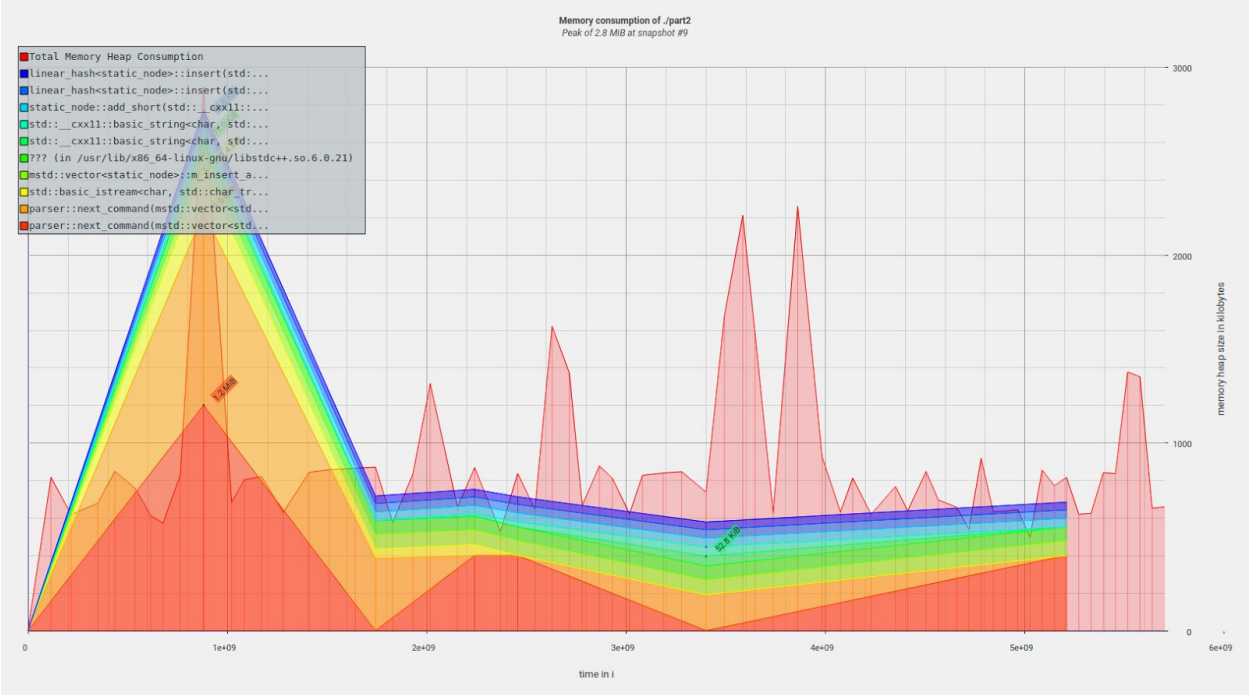
4. Χρήση Μνήμης

4.1 Part2

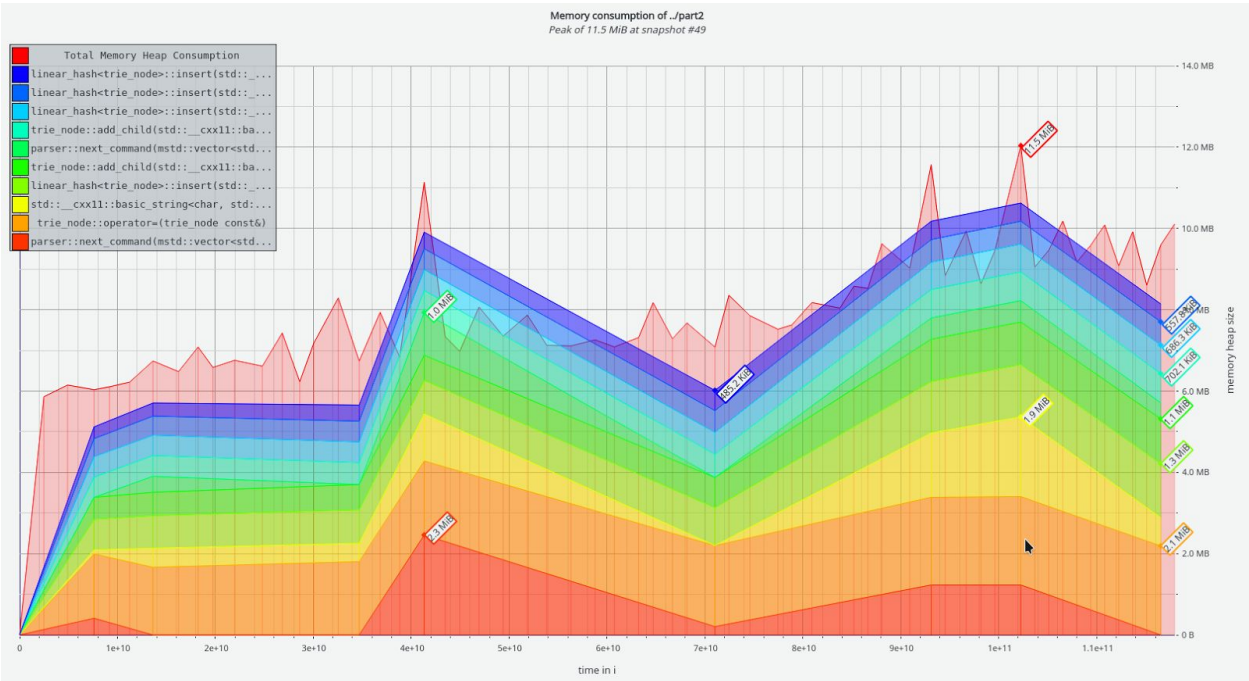
Small_Dynamic



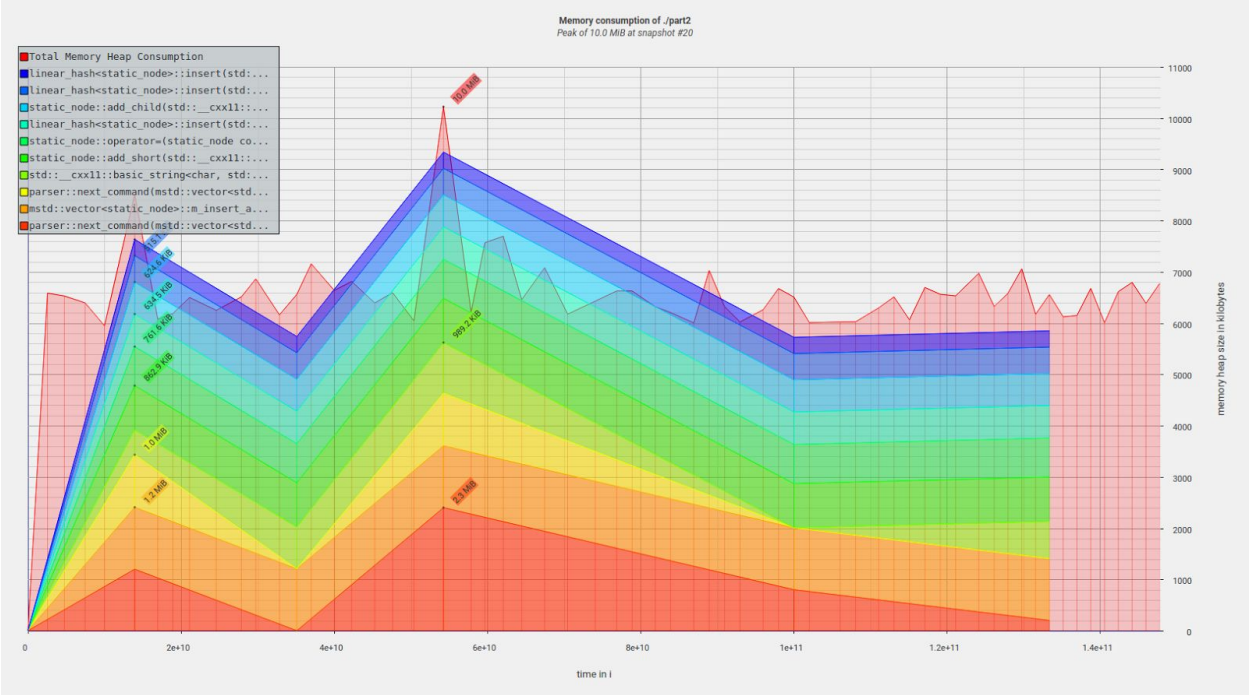
Small_Static



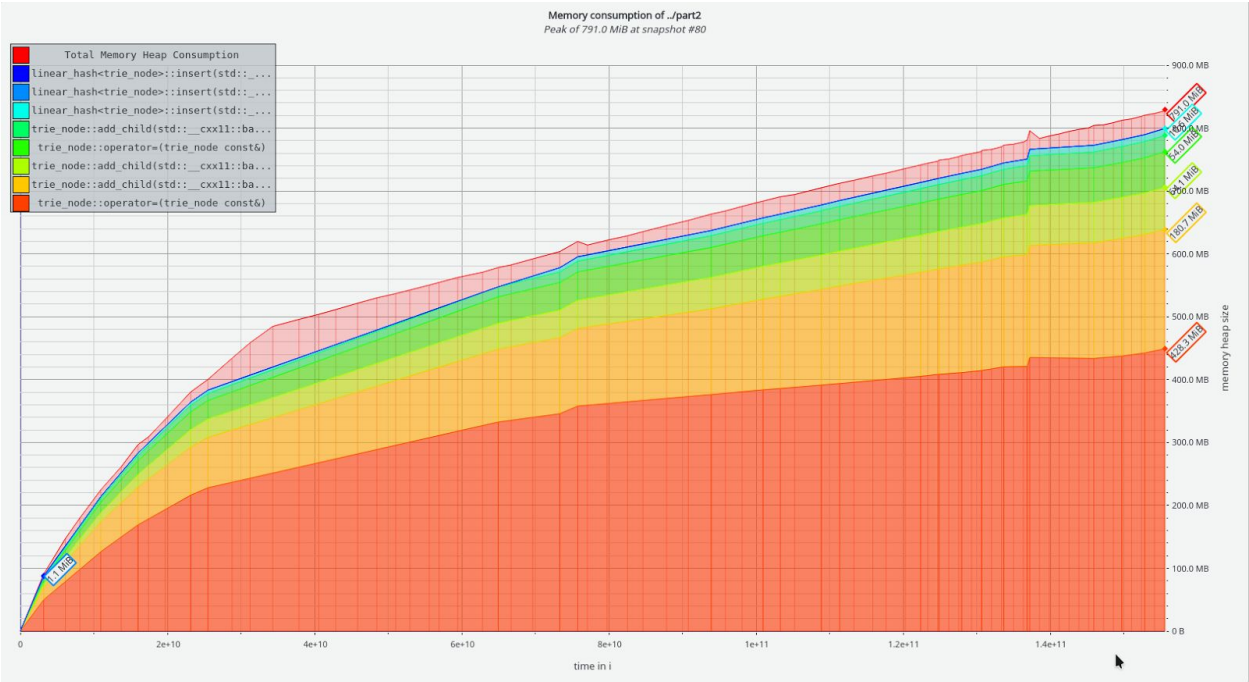
Medium_Dynamic



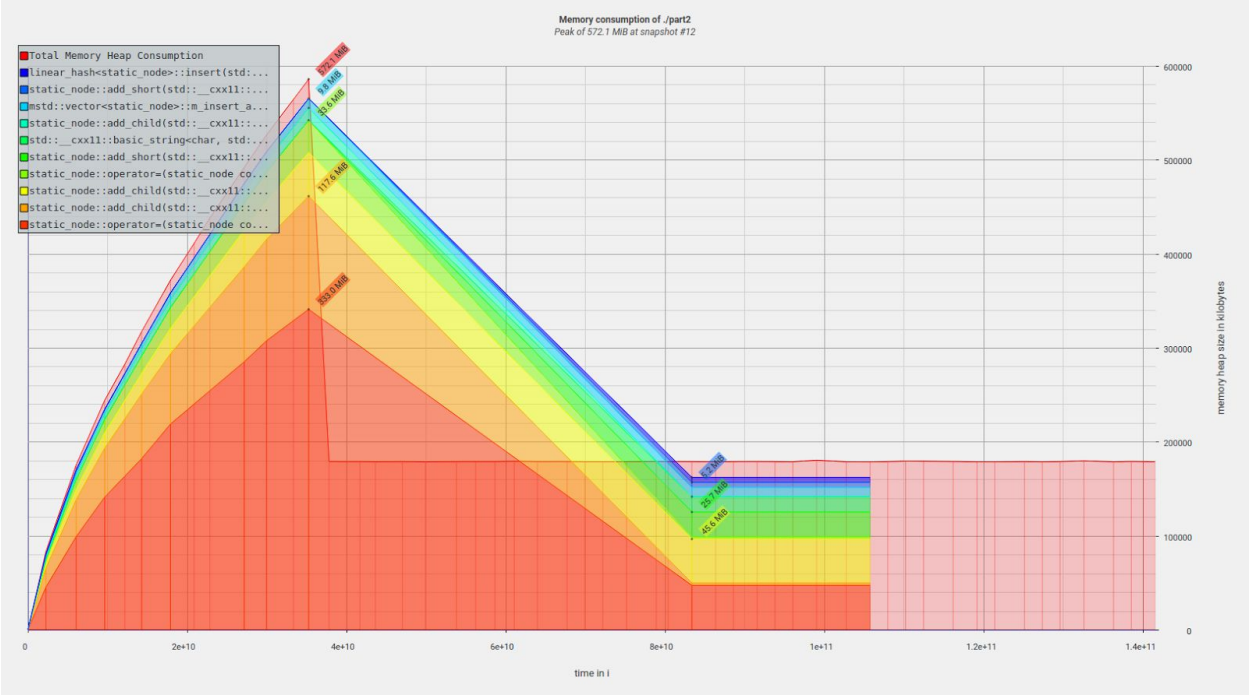
Medium_Static



Large_Dynamic

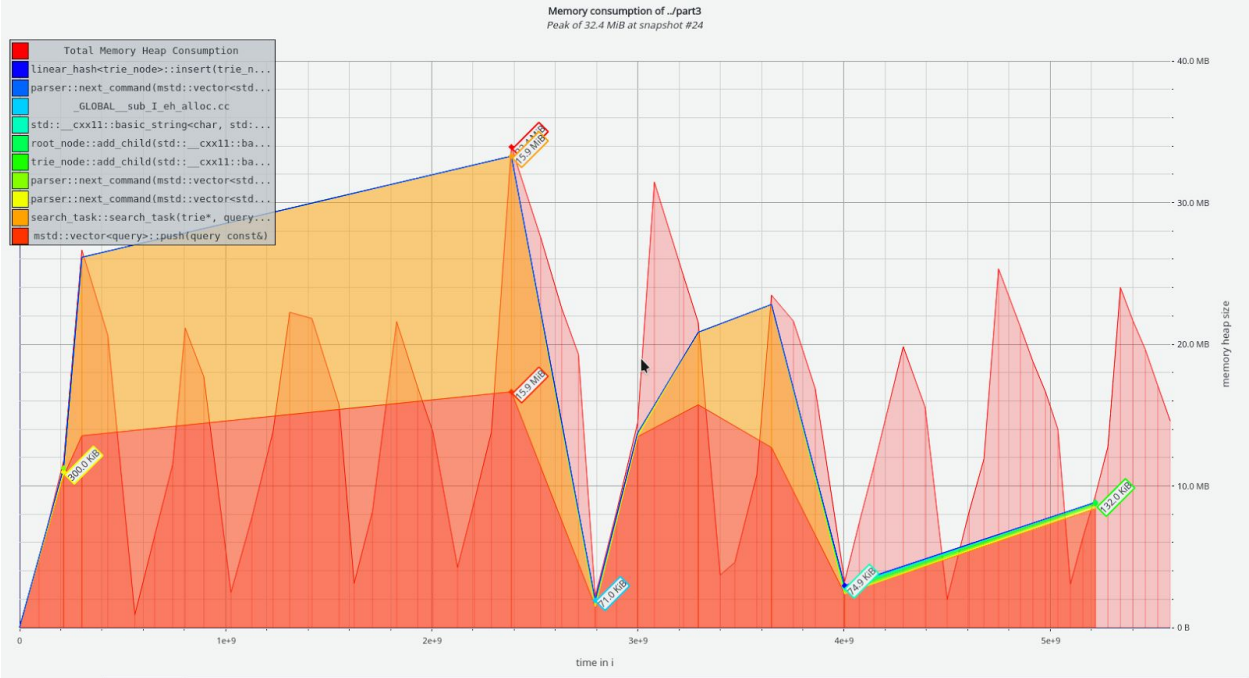


Large_Static

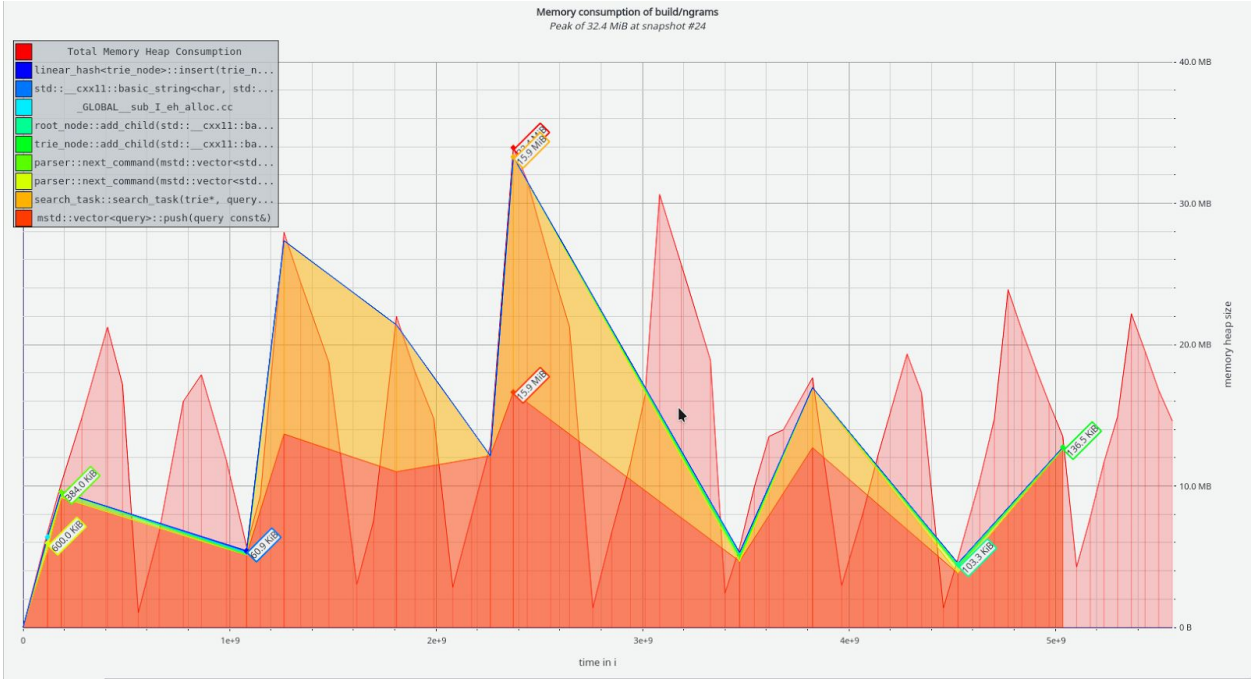


4.2 Part3

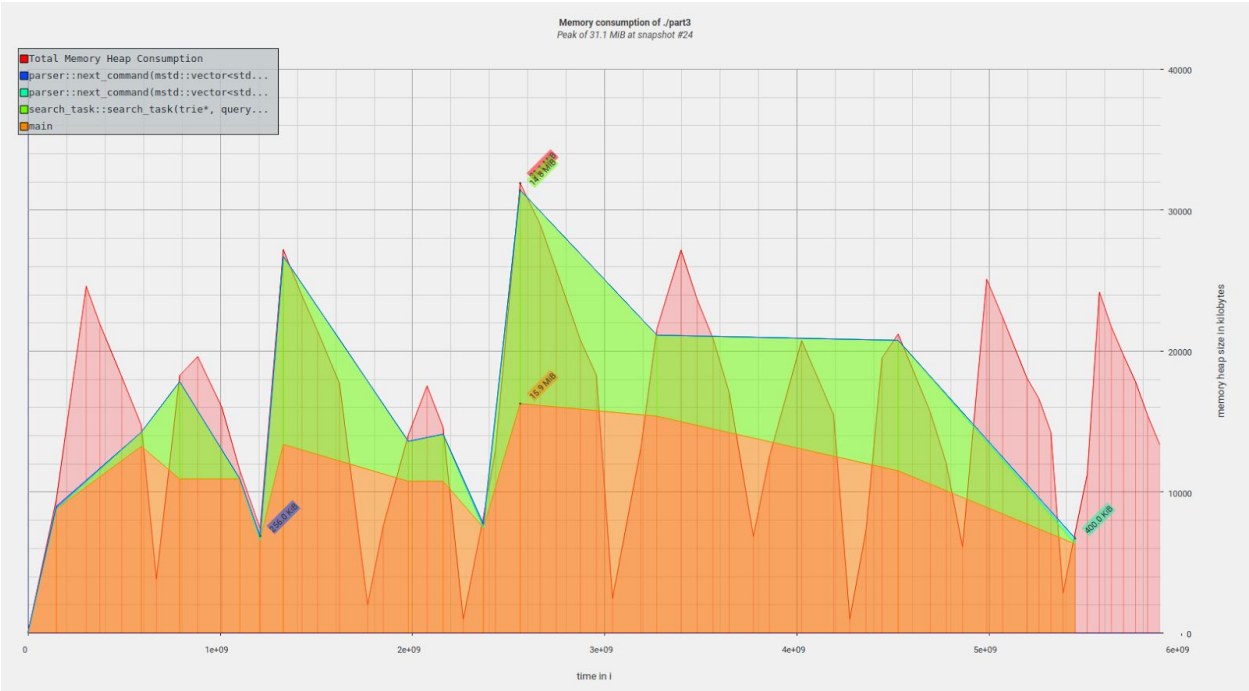
Small_Dynamic με clean



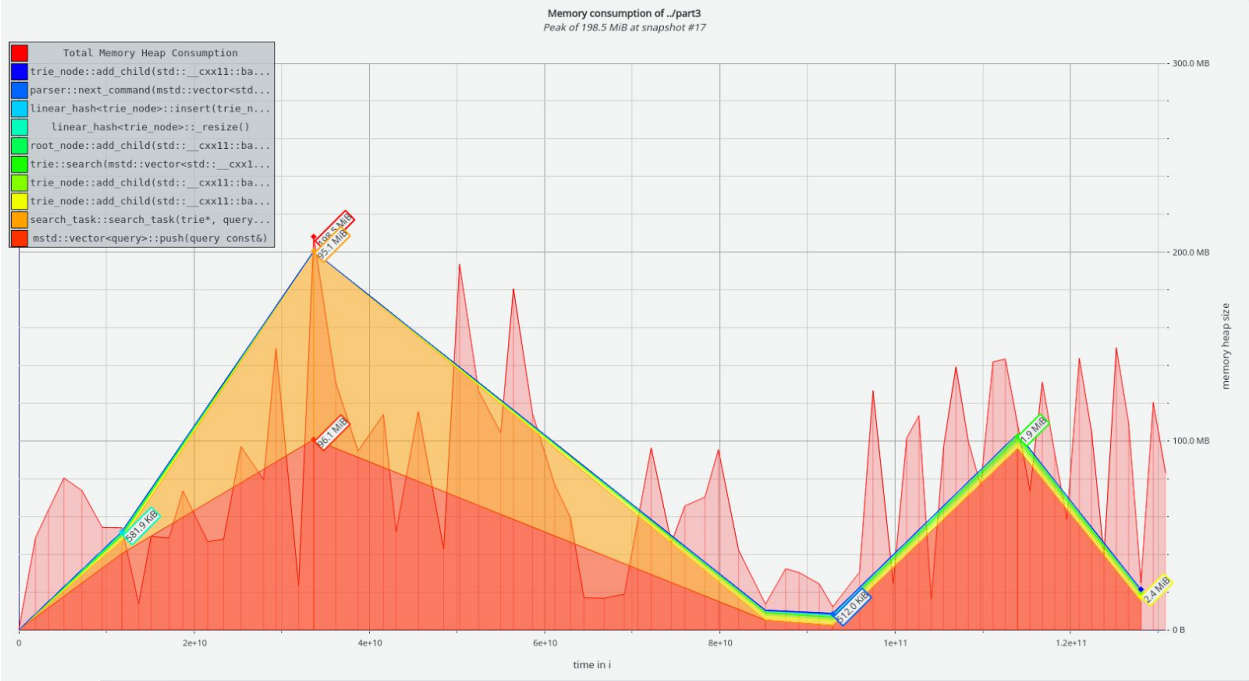
Small_Dynamic χωρίς clean up



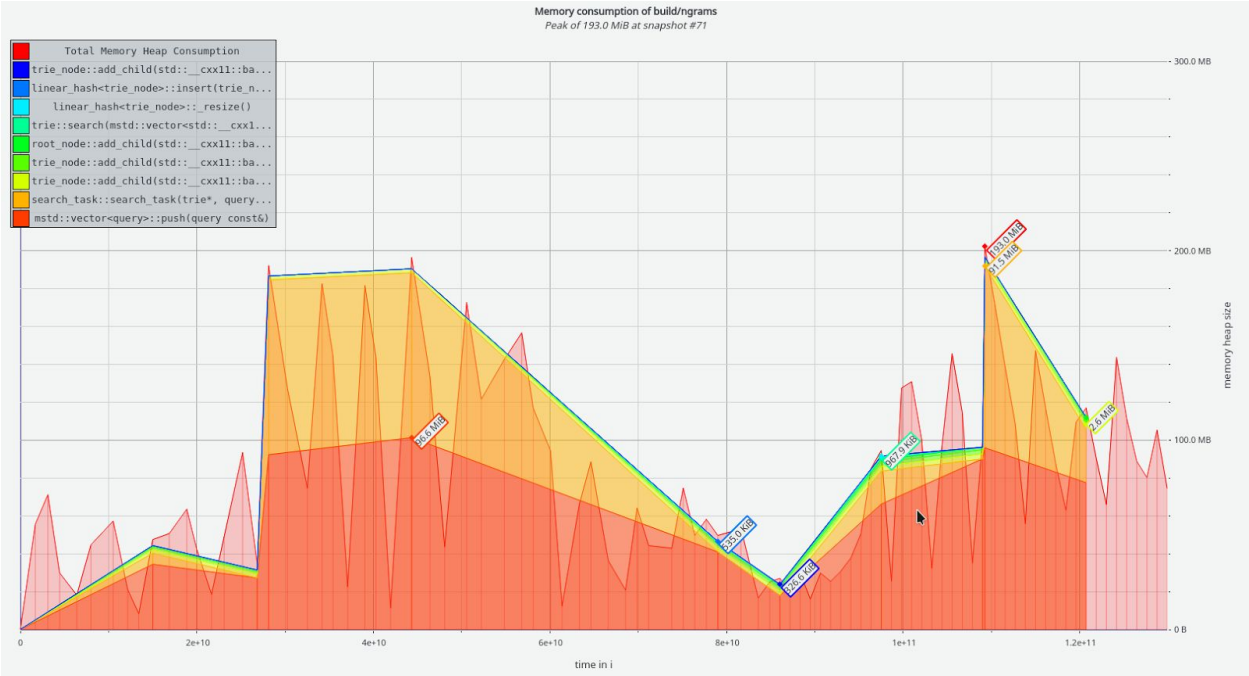
Small_Static



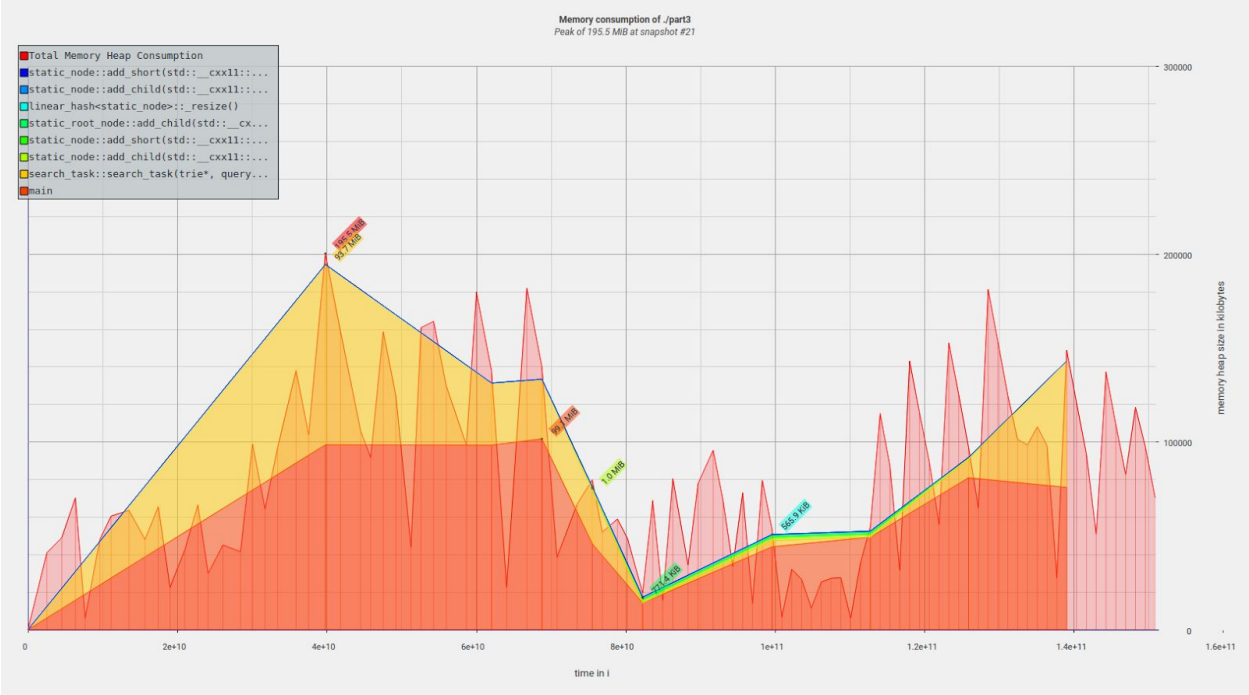
Medium_Dynamic με clean up



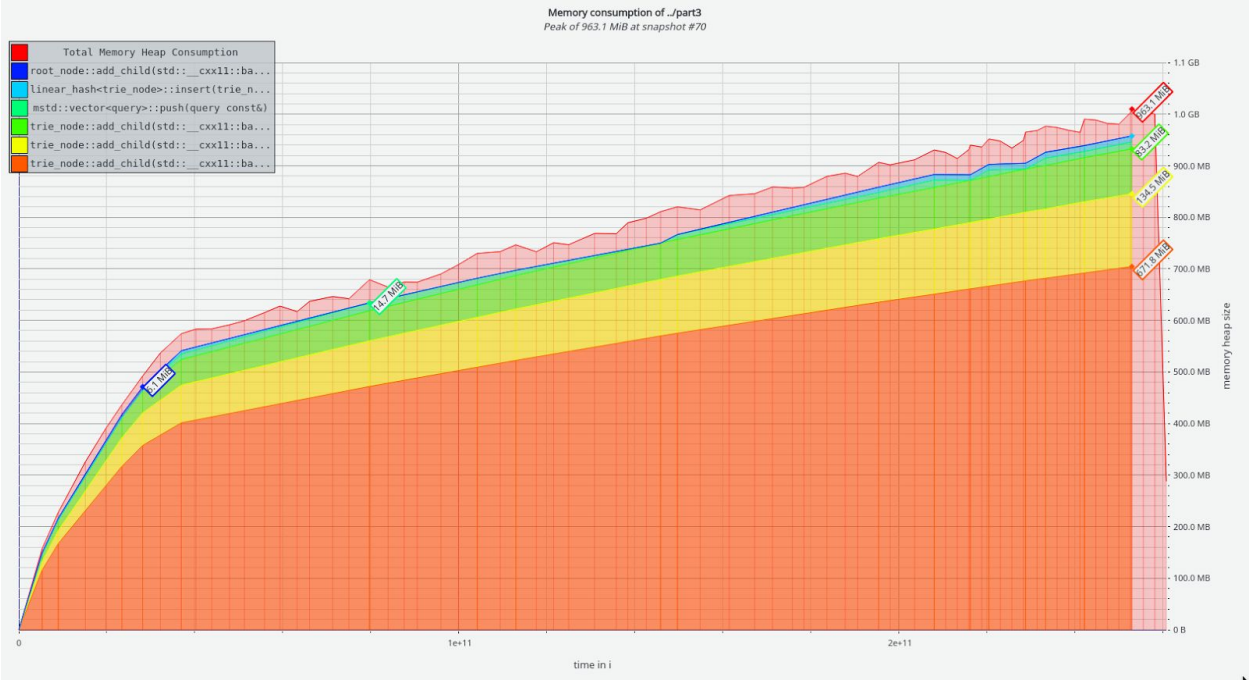
Medium_Dynamic χωρίς clean up



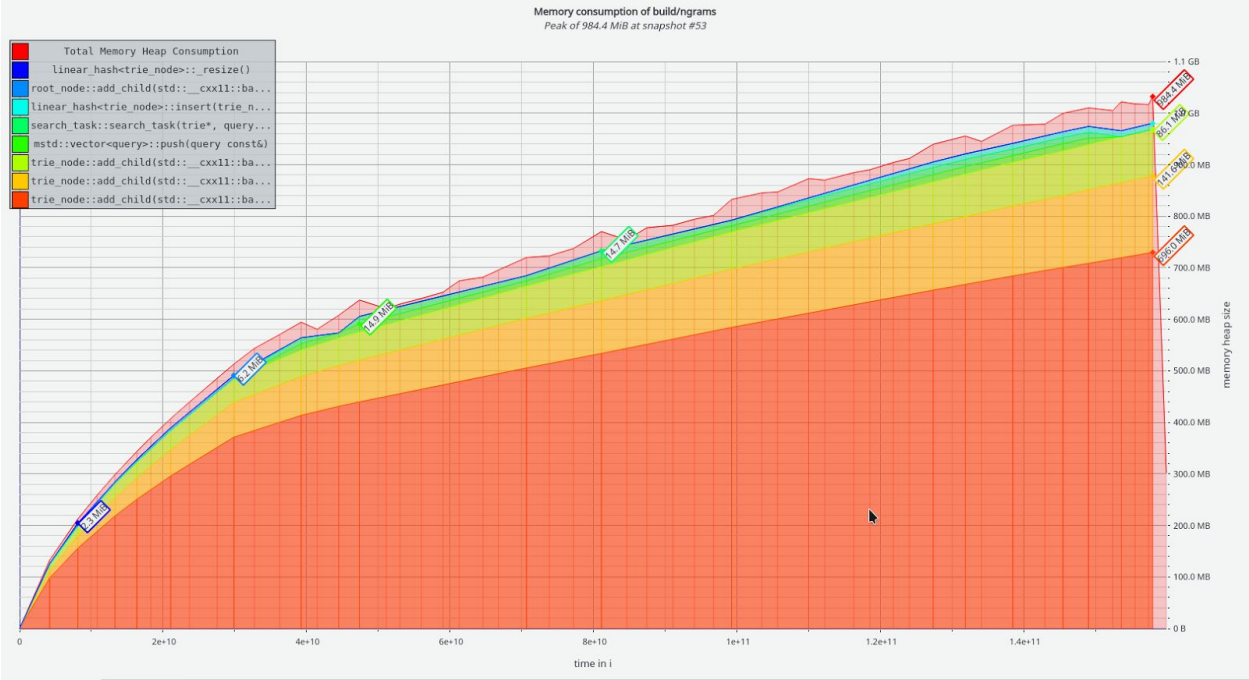
Medium_Static



Large_Dynamic με clean up



Large_Dynamic χωρίς clean up



Large_Static