Εργασία Εξαμήνου Δομές Δεδομένων

3868 Ζαρογιάννης Χριστόδουλος chrizaro@csd.auth.gr
3872 Τσιγγιρόπουλος Χρήστος Αλέξανδρος cdtsingi@csd.auth.gr

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ
2020-2021

Εισαγωγή:

Το πρόγραμμα συγγράφτηκε χρησιμοποιώντας το Code::Blocks 20.03 και δοκιμάστηκε με το Mingw.

main:

Μετά την δήλωση των βιβλιοθηκών και των απαραίτητων header files, ορίζουμε το μέγεθος του συνόλου Q. Στην συνέχεια ορίζουμε μεταβλητές για τις δομές μας, ανοίγουμε το αρχείο και εισάγουμε την πρώτη λέξη στις δομές μας. Αυτή η εισαγωγή γίνεται εκτός του βασικού βρόχου, που θα χρησιμοποιηθεί, ώστε να “αρχικοποιηθούν” οι δομές μας. Χρησιμοποιούμε μία **while (File >> s){}** για να παίρνουμε τις λέξεις από το αρχείο συνεχόμενα και μία συνάρτηση **clear(string)** (που βρίσκεται στο αρχείο clear.h) που καθαρίζει την λέξη από σημεία στίξης, κεφαλαία και νούμερα. Για παράδειγμα, η λέξη It’s θα γίνει its, η λέξη Master53!! θα γίνει master, η λέξη 56565 θα γίνει κενή συμβολοσειρά που δεν θα καταχωρηθεί. Οι 5 δομές και το σύνολο Q δημιουργούνται ταυτόχρονα. Για την εισαγωγή των στοιχείων καλούνται οι αντίστοιχες **insert()** της κάθε δομής. Για την insert() της κλάσης του AVL πρέπει να προσέξουμε ότι είναι απαραίτητα να αναθέτουμε την τιμή που γυρίζει στην ρίζα που έχουμε στην main, επειδή λόγω των περιστροφών της ισοστάθμισης μπορεί να αλλάξει ανά πάσα στιγμή. Τελειώσαμε με τις εισαγωγές, κλείνουμε το αρχείο που διαβάζαμε και ανοίγουμε το αρχείο που θα γράψουμε τα αποτελέσματα.

Ο κώδικάς που χρησιμοποιούμε για να εξάγουμε τα αποτελέσματα της αναζήτησης πάνω στο σύνολο Q, μας δίνει ένα αρχείο της μορφής:
x) όνομα_δομής:

λέξη1 αριθμός_εμφανίσεων_λέξης1
λέξη2 αριθμός_εμφανίσεων_λέξης2
λέξη1 αριθμός_εμφανίσεων_λέξης1
λέξη3 αριθμός_εμφανίσεων_λέξης3
λέξη4 αριθμός_εμφανίσεων_λέξης4

Elapsed time: y ms

Παρακάτω εξηγούνται οι εντολές που χρησιμοποιήθηκαν για την χρονομέτρηση.

Για την χρήση της διαγραφής, υπάρχει η εντολή remove και decrease για κάθε δομή. Στην περίπτωση των δύο δέντρων, η τιμή που επιστρέφει η συνάρτηση θα πρέπει εισάγεται στην ρίζα της δομής στην main, γιατί μπορεί να αλλάξει η ρίζα του δέντρου.

Αταξινόμητος πίνακας:

Για τα δεδομένα της κλάσης χρησιμοποιήθηκε ένα struct (το οποίο βρίσκεται υλοποιημένο στο struct.h), το οποίο έχει μία συμβολοσειρά και μία ακέραια μεταβλητή. Ο αρχικός πίνακας από struct είναι 1500 θέσεων.

Η λειτουργία της **αναζήτησης** στην κλάση του **αταξινόμητου πίνακα** υλοποιείται με την συνάρτηση **bool find(const string&, int&);** , η οποία δέχεται μία συμβολοσειρά με αναφορά και επιστρέφει true,αν βρήκε το στοιχείο, καθώς και την θέση του με αναφορά, και false, αν δεν το βρήκε, και γίνεται σειριακά με κόστος **O(n)**.

Η λειτουργία της **εισαγωγής** υλοποιείται με την συνάρτηση **void insert(const string&);** , η οποία εκτελεί αναζήτηση **bool find(const string&, int&);** και άμα υπάρχει η λέξη καλεί την **void increase(int);** να αυξήσει των αριθμό εμφανίσεων. Αν δεν βρεθεί η λέξη, τότε ελέγχεται αν υπάρχει χώρος στον πίνακα. Αν υπάρχει, τότε εισάγεται σε αυτόν. Αν δεν υπάρχει χώρος, τότε διπλασιάζεται ο πίνακας και μετά εισάγεται η λέξη. Το κόστος είναι **O(n)** λόγω της σειριακής αναζήτησης και δεν επηρεάζεται πολύ από την αύξηση ή όχι του πίνακα, O(2n)=O(n).

Για την λειτουργία της **διαγραφής** υπάρχουν δύο συναρτήσεις: μία που αφαιρεί εντελώς την λέξη από την δομή **void remove(const string&);** και μία που μειώνει τον αριθμό των εμφανίσεων της λέξης και αν φτάσουν 0, τότε την αφαιρεί από την δομή **void decrease(const string &);** . Και οι δύο πρώτα εκτελούν αναζήτηση, αν δεν βρουν την λέξη, δεν κάνουν τίποτα, αν την βρουν εκτελούν την λειτουργία τους. Όταν αφαιρείται μία λέξη από την δομή, το τελευταίο στοιχείο, τοποθετείται στην θέση αυτού που διαγράφηκε. Το κόστος είναι **O(n)** λόγω της σειριακής αναζήτησης.

Η κλάση του αταξινόμητου πίνακα επίσης περιέχει ένα κενό κατασκευαστή **arr();** , ένα καταστροφέα **~arr();** και ένα getter **int getT(int);** , ο οποίος εκτελεί σειριακή αναζήτηση με κόστος **O(n)** και επιστρέφει τις φορές που εμφανίστηκε μία λέξη στο αρχείο, αν δεν βρει την λέξη, επιστρέφει 0.

Ταξινομημένος πίνακας:

Ο ταξινομημένος πίνακας κληρονομεί από τον αταξινόμητο τον κατασκευαστή, τα δεδομένα και την συνάρτηση void increase(int); .

Η λειτουργία της **αναζήτησης** στην κλάση του **ταξινομημένου πίνακα** υλοποιείται με την συνάρτηση **bool find(const string&, int&);** , η οποία δέχεται μία συμβολοσειρά με αναφορά και επιστρέφει true,αν βρήκε το στοιχείο, καθώς και την θέση του με αναφορά, και false, αν δεν το βρήκε, και την συνάρτηση **int find2(word *b, const string &s, int low, int high);** , η οποία είναι υλοποιημένη αναδρομικά, καλείται μέσα στην **bool find(const string&, int&);** όταν δεν έχει βρει το στοιχείο και επιστρέφει την θέση στην οποία πρέπει να εισαχθεί το στοιχείο αν δεν υπάρχει. Η αναζήτηση είναι δυαδική με κόστος **O(logn)**.

Η λειτουργία της **εισαγωγής** υλοποιείται με την συνάρτηση **void insert(const string&);** , η οποία εκτελεί αναζήτηση **bool find(const string&, int&);** και άμα υπάρχει η λέξη καλεί την **void increase(int);** να αυξήσει των αριθμό εμφανίσεων. Αν δεν βρεθεί η λέξη, τότε ελέγχεται αν υπάρχει χώρος στον πίνακα. Αν υπάρχει, τότε μετακινούνται όλα τα στοιχεία που βρίσκονται από την θέση που πρέπει να εισαχθεί το στοιχείο και μετά, μία θέση πιο μετά και ύστερα εισάγεται σε αυτόν. Αν δεν υπάρχει χώρος, τότε διπλασιάζεται ο πίνακας (δημιουργείται νέος πίνακας με διπλάσιο χώρο και αντιγράφονται σε αυτόν τα στοιχεία) και μετά εισάγεται η λέξη. Το κόστος είναι **O(n + logn)** λόγω της δυαδικής αναζήτησης και της μετακίνησης των στοιχείων.

Για την λειτουργία της **διαγραφής** υπάρχουν δύο συναρτήσεις: μία που αφαιρεί εντελώς την λέξη από την δομή **void remove(const string&);** και μία που μειώνει τον αριθμό των εμφανίσεων της λέξης και αν φτάσουν 0, τότε την αφαιρεί από την δομή **void decrease(const string &);** . Και οι δύο πρώτα εκτελούν δυαδική αναζήτηση, αν δεν βρουν την λέξη, δεν κάνουν τίποτα, αν την βρουν εκτελούν την λειτουργία τους. Όταν αφαιρείται μία λέξη από την δομή, όλα τα στοιχεία που βρίσκονταν μία θέση μετά από τον στοιχείο που διαγράφτηκε, μετακινούνται μία θέση πριν για να καλύψουν το κενό που δημιουργήθηκε και να διατηρηθεί η διάταξη. Στην περίπτωση που αφαιρείται μία λέξη από την δομή, το κόστος είναι **O(n+logn)**, λόγω της σειριακής αναζήτησης και της μετακίνησης των στοιχείων. Στην περίπτωση μείωσης εμφανίσεων, είναι **O(logn)**, λόγω της δυαδικής αναζήτησης.

Η κλάση του ταξινομημένου πίνακα επίσης περιέχει ένα getter **int getT(int);** , ο οποίος εκτελεί δυαδική αναζήτηση με κόστος **O(logn)** και επιστρέφει τις φορές που εμφανίστηκε μία λέξη στο αρχείο, αν δεν βρει την λέξη, επιστρέφει 0.

Δυαδικό Δένδρο Αναζήτησης:

Έστω **h** το **ύψος** του δένδρου.

Η λειτουργία της **αναζήτησης** στην κλάση του **δυαδικού δένδρου αναζήτησης** υλοποιείται με την συνάρτηση **bool find(BST*, string);** , η οποία δέχεται την ρίζα του δέντρου και μία συμβολοσειρά και επιστρέφει true, αν βρήκε το στοιχείο, και false, αν δεν το βρήκε. Η αναζήτηση είναι δυαδική με μέσο κόστος **O(logh)** και χειρότερο κόστος **O(h)**.

Η λειτουργία της **εισαγωγής** υλοποιείται με την συνάρτηση **BST* insert(BST*, string);** , η οποία εκτελεί αναζήτηση **bool find(BST*, string);** και άμα υπάρχει η λέξη, της αυξάνει των αριθμό εμφανίσεων. Αν δεν βρεθεί η λέξη, την εισάγει στο δέντρο. Το μέσο κόστος είναι **O(logh)** και το χειρότερο κόστος **O(h)**, λόγω της δυαδικής αναζήτησης.

Για την λειτουργία της **διαγραφής** υπάρχουν δύο συναρτήσεις: μία που αφαιρεί εντελώς την λέξη από την δομή **BST* remove(BST*, string);** και μία που μειώνει τον αριθμό των εμφανίσεων της λέξης και αν φτάσουν 0, τότε την αφαιρεί από την δομή **BST* decrease(BST*, string);** . Και οι δύο πρώτα εκτελούν αναζήτηση, αν δεν βρουν την λέξη, επιστρέφουν την ρίζα που δέχτηκαν, αν την βρουν εκτελούν την λειτουργία τους και επιστρέφουν την ρίζα που πιθανώς να έχει αλλάξει. Το μέσο κόστος είναι **O(logh)** και το χειρότερο κόστος **O(h)**, λόγω της δυαδικής αναζήτησης.

Η κλάση του δυαδικού δένδρου αναζήτησης επίσης περιέχει ένα getter **int getT(BST*, string);** , ο οποίος εκτελεί δυαδική αναζήτηση και επιστρέφει τις φορές που εμφανίστηκε μία λέξη στο αρχείο, αν δεν βρει την λέξη, επιστρέφει 0 και δύο κατασκευαστές, **BST();** , **BST(string);**

Τέλος υπάρχουν και 3 συναρτήσεις διάσχισης του δέντρου.

Η **void inorder(BST*, ofstream&);** εξάγει σε μία μεταβλητή ofstream τα στοιχεία του δέντρου ταξινομημένα χρησιμοποιώντας ενδοδιατεταγμένη διάσχιση.

Η **void preorder(BST*, ofstream&);** εξάγει σε μία μεταβλητή ofstream τα στοιχεία του δέντρου χρησιμοποιώντας προδιατεταγμένη διάσχιση.

Η **void postorder(BST*, ofstream&);** εξάγει σε μία μεταβλητή ofstream τα στοιχεία του δέντρου χρησιμοποιώντας μεταδιατεταγμένη διάσχιση.

Δυαδικό Δένδρο Αναζήτησης AVL:

Η κλάση του δυαδικού δένδρου αναζήτησης AVL υλοποιήθηκε ως παράγωγη κλάση του δυαδικού, αλλά τα μόνο στοιχεία που κληρονόμησε είναι κάποια δεδομένα.

Έστω **h** το **ύψος** του δέντρου.

Η λειτουργία της **αναζήτησης** στην κλάση του **δυαδικού δένδρου αναζήτησης AVL** υλοποιείται με την συνάρτηση **bool find(AVL*, string);** , η οποία δέχεται την ρίζα του δέντρου και μία συμβολοσειρά και επιστρέφει true, αν βρήκε το στοιχείο, και false, αν δεν το βρήκε. Η αναζήτηση είναι δυαδική με κόστος **O(logh)**.

Η λειτουργία της **εισαγωγής** υλοποιείται με την συνάρτηση **AVL* insert(AVL*, string);** , η οποία εκτελεί αναζήτηση **bool find(AVL*, string);** και άμα υπάρχει η λέξη, της αυξάνει των αριθμό εμφανίσεων. Αν δεν βρεθεί η λέξη, την εισάγει στο δέντρο. Το κόστος είναι **O(logh)** λόγω της δυαδικής αναζήτησης. Μετά την εισαγωγή, ενημερώνεται το ύψος του κόμβου χρησιμοποιώντας τον getter **int getHeight(AVL*);** , η οποία επιστρέφει το ύψος του κόμβου και την συνάρτηση **int max(int, int);** , η οποία συγκρίνει δύο αριθμούς και επιστρέφει τον μέγιστο. Στην συνέχεια ελέγχει αν το δέντρο παραμένει ισοζυγισμένο με την συνάρτηση **int getBalance(AVL*);** , η οποία επιστρέφει την διαφορά υψών και αν η απόλυτη τιμή της διαφοράς είναι μεγαλύτερη από 1, τότε εκτελούνται κατάλληλες περιστροφές για να ισορροπήσει το δέντρο. Η συνάρτηση **AVL* rightRotate(AVL*);** , υλοποιεί την δεξιά περιστροφή και η **AVL* leftRotate(AVL*);** , την αριστερή περιστροφή. Το κόστος της εισαγωγής είναι **O(logh)**.

Για την λειτουργία της **διαγραφής** υπάρχουν δύο συναρτήσεις: μία που αφαιρεί εντελώς την λέξη από την δομή **AVL* remove(AVL*, string);** και μία που μειώνει τον αριθμό των εμφανίσεων της λέξης και αν φτάσουν 0, τότε την αφαιρεί από την δομή **AVL* decrease(AVL*, string);** . Και οι δύο πρώτα εκτελούν αναζήτηση, αν δεν βρουν την λέξη, επιστρέφουν την ρίζα που δέχτηκαν, αν την βρουν εκτελούν την λειτουργία τους και επιστρέφουν την ρίζα που πιθανώς να έχει αλλάξει. Αφού αφαιρεθεί ο κόμβος, ελέγχεται αν το δέντρο είναι ισορροπημένο και αν δεν είναι εκτελούνται οι κατάλληλες περιστροφές. Το κόστος είναι **O(logh)**. Στην υλοποίηση της διαγραφής χρησιμοποιείται και η συνάρτηση **AVL* minValue(AVL*);** , που στην περίπτωση που διαγράφουμε κόμβο με δύο παιδιά, βρίσκει ποιος κόμβος θα μπει στην θέση του. Το κόστος είναι **O(logh)**.

Η κλάση του **δυαδικού δένδρου αναζήτησης AVL** επίσης περιέχει ένα getter **int getT(AVL*, string);** , ο οποίος εκτελεί δυαδική αναζήτηση και επιστρέφει τις φορές που εμφανίστηκε μία λέξη στο αρχείο, αν δεν βρει την λέξη, επιστρέφει 0 και δύο κατασκευαστές **AVL();** , **AVL(string);**

Τέλος υπάρχουν και 3 συναρτήσεις διάσχισης του δέντρου.
Η **void inorder(AVL*, ofstream&);** εξάγει σε μία μεταβλητή ofstream τα στοιχεία του δέντρου ταξινομημένα χρησιμοποιώντας ενδοδιατεταγμένη διάσχιση.
Η **void preorder(AVL*, ofstream&);** εξάγει σε μία μεταβλητή ofstream τα στοιχεία του δέντρου χρησιμοποιώντας προδιατεταγμένη διάσχιση.
Η **void postorder(AVL*, ofstream&);** εξάγει σε μία μεταβλητή ofstream τα στοιχεία του δέντρου χρησιμοποιώντας μεταδιατεταγμένη διάσχιση.

Πίνακας Κατακερματισμού με ανοιχτή διεύθυνση:

Με λίγα λόγια αυτή η δομή αποθηκεύει σε μια συγκεκριμένη θέση του πίνακα ,στοιχεία μιας συμβολοσειράς. Η θέση αυτή, είναι μοναδική για κάθε διαφορετική τιμή, αφού παράγεται από μια **συνάρτηση Κατακερματισμού,** που παίρνει ως όρισμα την τιμή αυτή.Σε περίπτωση σύγκρουσης, για δύο διαφορετικές τιμές, δημιουργείται μια νέα θέση. Ειδικότερα :

Αρχικά ο πίνακας κατακερματισμού δημιουργείται με 1500 θέσεις όπου κάθε θέση ανήκει στην κλάση **ht** η οποία αποτελείται από 3 στοιχεία :

- Μία μεταβλητή **enable** τυπου **bool** που αποθηκεύει αν έχει αποθηκευτεί στοιχείο ή όχι (αρχικοποίηση σε false)
- Μία μεταβλητή **w** τύπου **string** που αποθηκεύει την συμβολοσειρά
  και
- Μία μεταβλητή **t** τύπου **int** που αποθηκεύει το πόσες φορές έχει εμφανιστεί αυτή η συμβολοσειρά (αρχικοποίηση σε 0)

Η λειτουργία της **αναζήτησης** στην κλάση του **Πίνακα Κατακερματισμού** υλοποιείται με τη συνάρτηση **bool find(string, int& i);** , η οποία δέχεται μία συμβολοσειρά και επιστρέφει true , αν βρήκε το στοιχείο, καθώς και την θέση **i** της συμβολοσειράς που γυρνάει με αναφορά, και false, αν δεν το βρήκε, και γίνεται με κόστος **O(1)** αφού η θέση βγαίνει από την **Συνάρτηση Κατακερματισμού** .

Η λειτουργία της **εισαγωγής** υλοποιείται με την συνάρτηση **void insert(string);** , η οποία εκτελεί αναζήτηση **bool find(string s, int& i);** και άμα υπάρχει η λέξη s αυξάνει τον αριθμό εμφανίσεων της ίδιας κατά μία μονάδα (όπου η s βρίσκεται στην θέση i του πίνακα) .
Αν δεν βρεθεί η λέξη s, τότε η i μεταβλητή , που γύρισε αναφορικά, θα έχει την θέση του πίνακα που θα έπρεπε να αποθηκευτεί η s και άρα στην θέση αυτή του πίνακα αποθηκεύεται στην μεταβλητή : **w** το **s , t** το **1** και στην **enable** το **true**. Με κόστος μέχρι στιγμή **O(1).**
Στην συνέχεια ελέγχεται αν έχει γεμίσει παραπάνω από το μισό του πίνακα για να αυξήσουμε τον πίνακα κατακερματισμου.Αν όχι τότε δεν διπλασι Αν ναι, τότε διπλασιάζεται ο πίνακας και μετά τοποθετούμε τα στοιχεία (μόνο αυτά με **enable=true**) από τον παλιό πίνακα στον νέο , σειριακά περνώντας τα πρώτα από την **συνάρτηση κατακερματισμού** .Το κόστος του διπλασιασμού είναι **O(n)**.
Άρα το κόστος εισαγωγής είναι **O(1)** χωρίς αλλαγή του πίνακα και επηρεάζεται πολύ από την αύξηση του πίνακα καθώς τότε η εισαγωγή γίνεται σε **O(n)**.

Η λειτουργία της **Συνάρτηση Κατακερματισμού** υλοποιείται με την συνάρτηση **int F(string s,int a);** , η οποία επιστρέφει την θέση που θα έπρεπε να αποθηκευτεί ή είναι η συμβολοσειρά **s.** Ειδικότερα γίνονται πράξεις με: την τιμή **α** (με α = σύνολο των συγκρούσεων) , με πρώτους αριθμούς , με τα γράμματα που έχει κάθε λέξη καθώς και με την θέση του κάθε γράμματος , στοιχεία που καθιστούν την τιμή επιστροφής αρκετα μοναδική και περίπλοκη . Πριν γίνει η επιστροφή το αποτέλεσμα περιορίζεται μεταξύ του 0 και του size-1 του πίνακα (δλδ στις θέσεις του πίνακα) μέσα από την πράξη x MOD size όπου x το αποτέλεσμα των παραπάνω πράξεων.

Τέλος υπάρχουν και 4 συναρτήσεις getters:

1. GetTimes(int i);
   Που επιστρέφει το t δλδ τις εμφανίσεις της μεταβλητής w της θέσης i

2. GetWord(int i);
   Που επιστρέφει το w δλδ την συμβολοσειρά που έχει αποθηκευτεί στην θέση i

3. GetSize();
   Που επιστρέφει το μέγεθος που καταλαμβάνει ο πίνακας

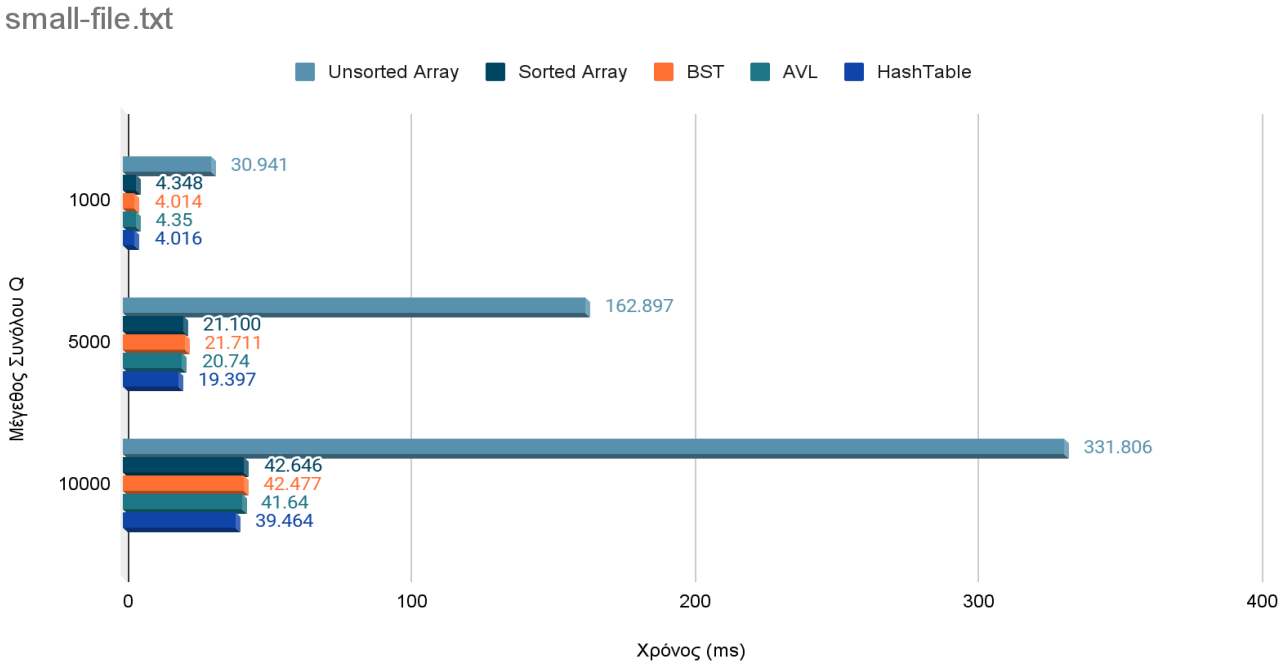
4. GetEnabled(int i);
   Που επιστρέφει αν έχει εισαχθεί στοιχείο στην θέση i του πίνακα κατακερματισμού

Χρονομέτρηση Δομών:

Για την χρονομέτρηση της αναζήτησης επί των δομών χρησιμοποιήθηκε η βιβλιοθήκη **<chrono>** και οι εντολές
auto start = chrono::high_resolution_clock::now();
auto finish = chrono::high_resolution_clock::now();
chrono::duration<double> elapsed = finish - start;
out<<endl<<"Elapsed time: "<<elapsed.count()<<"s"<<endl<<endl;

Για την επιλογή των στοιχείων του συνόλου Q επιλέχθηκαν οι τελευταίες λέξεις του αρχείου ώστε να ζοριστούν το περισσότερο δυνατόν. Για να γίνει αυτό, ουσιαστικά ο πίνακας είναι “κυκλικός” και επανεγγράφονται λέξεις πάνω σε άλλες. Υπάρχει και μία μεταβλητή **stop** η οποία σταματάει να αυξάνεται αν φτάσουμε το μέγεθος του συνόλου και την χρησιμοποιούμε για την συνθήκη στην if της εκτύπωσης ώστε να μην εκτυπωθούν σκουπίδια σε περίπτωση μικρού αρχείου.

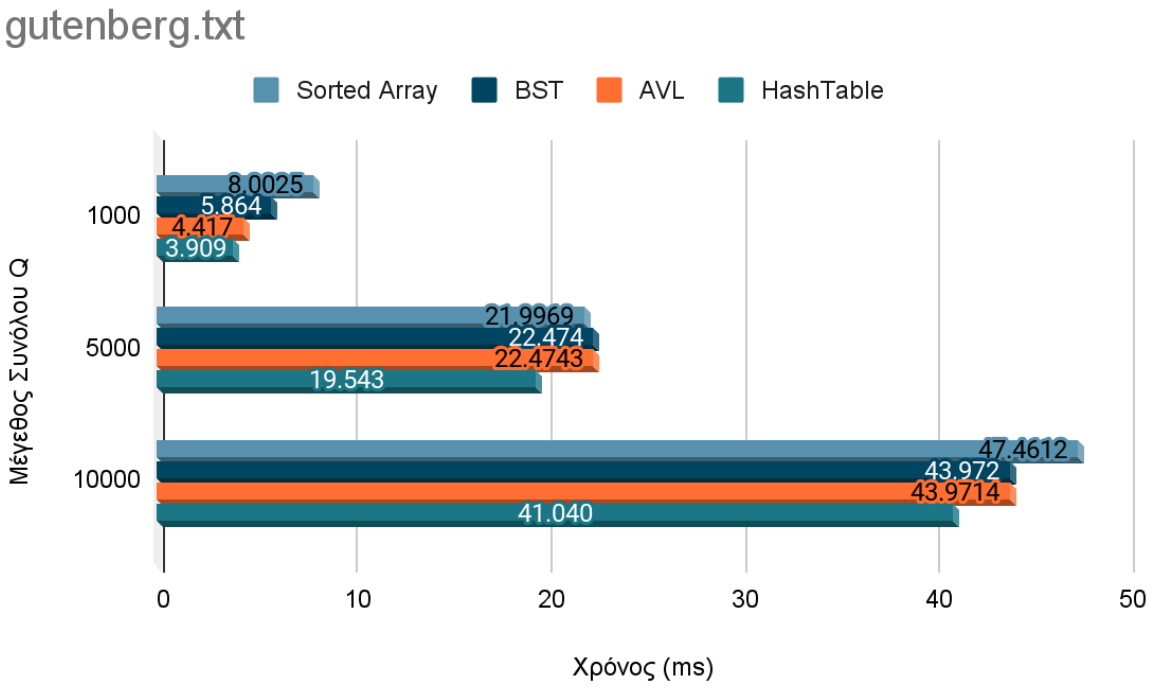
Το αρχείο small-file.txt έδωσε αυτούς τους χρόνους:
Κάθε δομή χρονομετρήθηκε 3 φορές και παρακάτω αναφέρεται ο μέσος όρος των μετρήσεων.



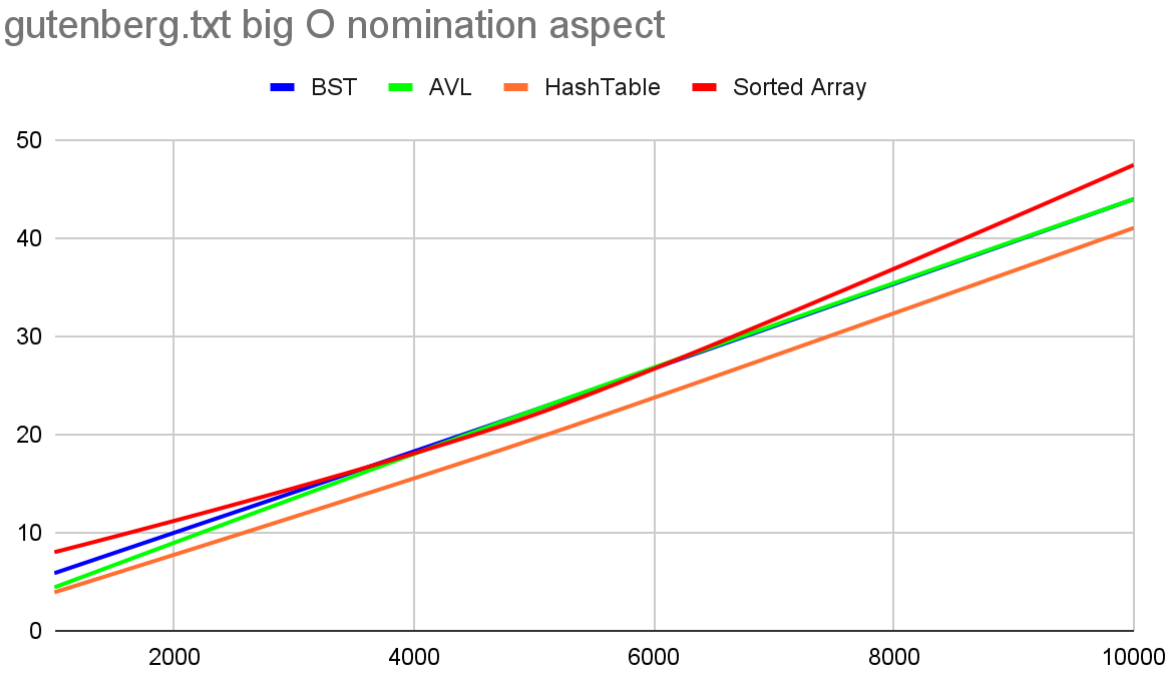
Από τα παραπάνω γίνεται αντιληπτό ότι το unsorted Array είναι μια από τις χειρότερες δομές για να χρησιμοποιήσουμε αφού η αναζήτηση κάνει περίπου 8 φορές περισσότερο χρόνο από τις υπόλοιπες τέσσερις άσχετα με το πλήθος των λέξεων. Επιπλέον, παρατηρούμε ότι όσο μεγαλώνει ο αριθμός τον λέξεων στην αναζήτηση (5000->10000), τόσο μεγαλώνουν οι διαφορές στους χρόνους των διαφορετικών δομών, με τους μικρότερους χρόνους να τους πετυχαίνει ο Πίνακας Κατακερματισμού (για μεγαλύτερα πλήθη λέξεων). Όσον αφορά τους χρόνους για λιγότερες λέξεις(<=1000), βλέπουμε ότι για την αναζήτηση η καλύτερη δομή είναι πάλι ο Πίνακας Κατακερματισμού μαζί με το Δυαδικό Δέντρο Αναζήτησης με πολύ

μικρές διαφορές με τις άλλες δομες, στοιχείο που μας φαίνεται λογικό αφού δεν προλαβαίνουν να εμφανιστούν τα προτερήματα τις κάθε δομής στην αναζήτηση . Συμπερασματικά , παρατηρούμε ότι όσο μεγαλύτερο το σύνολο Q τόσο πιο πολύ αυξάνεται η διαφορά του Πίνακα Κατακερματισμού με το δένδρο AVL που είναι επίσης αρκετά γρήγορο στην αναζήτηση σε μεγάλο πλήθος.

Το αρχείο **gutenberg.txt** έδωσε αυτούς τους χρόνους:
Για το αρχείο gutenberg.txt δεν χρονομετρήθηκε η δομή του αταξινόμητου πίνακα, καθώς δεν μπορούσε να γίνει σε εύλογο χρονικό διάστημα, λόγω μεγάλου όγκου δεδομένων και αργής δομής, συμπέρασμα που εξάγεται και από τις μετρήσεις στο προηγούμενο αρχείο.



Παρατηρούμε ότι ο πίνακας κατακερματισμού είναι η ταχύτερη δομή. Τα δύο δυαδικά δέντρα είναι πολύ κοντά, αλλά σε μικρό σύνολο Q μπορούμε να δούμε ότι το AVL είναι γρηγορότερο από το απλό. Η δομή του ταξινομημένου πίνακα είναι αρκετά γρήγορη στην αναζήτηση, αλλά παραμένει στην τελευταία θέση ειδικά για μεγάλο Q.
Για τις δομές των δέντρων και του hash table, ο χρόνος που χρειάζεται είναι σχεδόν ανάλογος του μεγέθους, πχ για τον hash table για Q=5000 είναι περίπου 20ms και για διπλάσιο Q (10000) είναι περίπου 40ms. Για AVL Q=1000, ο χρόνος είναι 4,4ms και για πενταπλάσιο Q (5000), ο χρόνος είναι σχεδόν πενταπλάσιος 22ms.
Ο ταξινομημένος πίνακας, από την άλλη, δεν εμφανίζει τέτοια αναλογία.



Χρόνοι για την λειτουργία της εισαγωγής και της εκτέλεσης:
Στην περίπτωση του small-file.txt ο χρόνος εκτέλεσης του προγράμματος είναι περίπου 4,8s.

Στην περίπτωση του gutenberg.txt, ο χρόνος εκτέλεσης του προγράμματος για τα δύο δέντρα και τον hash table είναι περίπου 490s για Q 10000 στοιχείων, 475s για Q 5000 στοιχείων και 493s για Q 1000. Ο χρόνος κατασκευής της δομή του δέντρου (οποιουδήποτε) είναι περίπου 200s. Ο χρόνος κατασκευής της δομή του hashtable είναι περίπου 100s.

Στην περίπτωση του gutenberg.txt, ο χρόνος εκτέλεσης του προγράμματος για τον ταξινομημένο πίνακα είναι 7,5 ώρες.