Εργασία 2020
Δομές Δεδομένων
Ονοματεπώνυμο: Νικόλαος Παπαγερούδης
ΑΕΜ: 2918
15 Ιουνίου 2020

Εισαγωγή

Το πρόγραμμα αποτελείται από 6 κλάσεις και την main. Πιο συγκεκριμένα για κάθε μία από τις δομές **Binary Search Tree**, **AVL Tree** και **Hash Table** υπάρχουν 2 κλάσεις. Μία που αναπαριστά το εκάστοτε δένδρο ή τον πίνακα κατακερματισμού, και μία που αναπαριστά τους κόμβους του δένδρων ή τα κελιά του πίνακα αντίστοιχα.

Όλες οι τιμές διαβάζονται από το αρχείο input.txt που υπάρχει στον φάκελο Project του προγράμματος. Τα αποτελέσματα της εκτέλεσης του προγράμματος εμφανίζονται στην οθόνη, αλλά αποθηκεύονται και στο αρχείο output.txt που βρίσκεται επίσης στον φάκελο Project του προγράμματος.

Έγκυρη λέξη θεωρείται αυτή που έχει τουλάχιστον 1 χαρακτήρα (A-Z, a-z). Οπότε το πρόγραμμα αγνοεί όλα τα σύμβολα και αποθηκεύει ξεχωριστά τις λέξεις που διαχωρίζονται με σύμβολα. Για παράδειγμα αν στο κείμενο υπάρχει το email: *test@csd.auth.gr* θα αποθηκευτούν οι λέξεις: *test, csd, auth, gr*. Επίσης δεν έχει σημασία εάν μια λέξη χρησιμοποιεί πεζά ή κεφαλαία γράμματα. Η λέξη *Test* θεωρείται ίση με τη λέξη *test*, γι' αυτό το λόγο όλα τα κεφαλαία γράμματα των λέξεων μετατρέπονται σε μικρά. Ο έλεγχος της εγκυρότητας κάθε χαρακτήρα γίνεται με τη χρήση του κώδικα *ASCII*

Ο αριθμός **Q** των λέξεων που είναι προς αναζήτηση ορίζεται στο πάνω μέρος της main.cpp ως: #define Q 10.000. Στο παράδειγμα το **Q** ορίζεται ως *10.000* αλλά η τιμή του μπορεί να αλλάξει. Ο τρόπος που επιλέγονται οι λέξεις είναι ο εξής: Αρχικά ορίζεται ο αριθμός ***Q***, 10.000 στο παράδειγμά μας. Έπειτα ο αριθμός P = 246.848/Q ο οποίος είναι ο μέγιστος επιτρεπτός που μπορεί να χρησιμοποιηθεί εάν θέλουμε να αποθηκεύουμε μία για κάθε **Q** λέξεις. Σε συνδυασμό με τη συνάρτηση rand() πρώτη τυχαία λέξη θα είναι η i-οστή που διαβάστηκε από το input.txt (όπου $0 \le i \le P$). Η δεύτερη λέξη που θα επιλεγεί θα είναι η i2-οστή λέξη που διαβάστηκε από το αρχείο (όπου $i < i2 \le P$) κ.ο.κ. Ο αριθμός *246.848* που χρησιμοποιείται για τον προσδιορισμό του P είναι ο συνολικός αριθμός των λέξεων που εισάγονται.

Binary Search Tree

Η δομή **Binary Search Tree** αποτελείται από τις κλάσεις:

α) **NodeBST:**

Η κλάση *NodeBST* αναπαριστά τους κόμβους του δυαδικού δένδρου. Γι' αυτό και χρησιμοποιούνται οι μεταβλητές:

- **string word**, που αναπαριστά την λέξη του συγκεκριμένου κόμβου
- **int count**, η οποία μετράει τις φορές που εισήλθε στο δένδρο η λέξη
- **NodeBST* left**, που αναπαριστά το αριστερό παιδί του κόμβου
- **NodeBST* right**, που αναπαριστά το δεξί παιδί του κόμβου

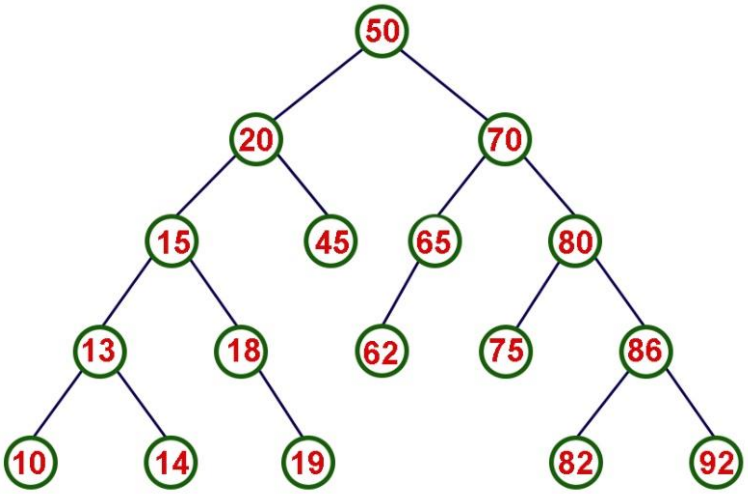
β) **BinarySearchTree**:

Η κλάση **BinarySearchTree** αποτελείται από έναν κενό κατασκευαστή και 6 μεθόδους, μία για κάθε λειτουργία του δένδρου που ζητήθηκε από την εκφώνηση (*εισαγωγή, διαγραφή, αναζήτηση, inorder, preorder, postorder*). Αναλυτικότερα:

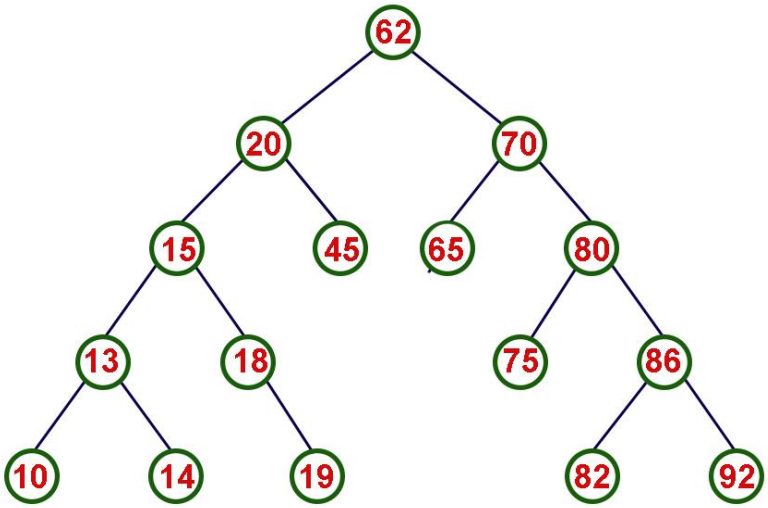
- **NodeBST* insertWord**: Η συγκεκριμένη μέθοδος αναπαριστά την **εισαγωγή** νέας λέξης στο δυαδικό δένδρο αναζήτησης. Έχει ως ορίσματα i) τη ρίζα του δένδρου (εάν αυτή υπάρχει αλλιώς ισούται με **nullptr**) και ii) την λέξη που θα εισαχθεί στο δένδρο. Η μέθοδος είναι αναδρομική. Κάθε φορά που καλείται ελέγχει εάν η λέξη που πρόκειται να εισαχθεί βρίσκεται λεξικογραφικά πριν ή μετά από τη λέξη του κόμβου που δόθηκε ως όρισμα. Εάν βρίσκεται πριν τότε καλείται ξανά η μέθοδος έχοντας ως όρισμα το **αριστερό παιδί** του κόμβου (node->left) στο οποίο και εισάγεται η τιμή. Εάν βρίσκεται μετά τότε καλείται ξανά η μέθοδος έχοντας ως όρισμα το **δεξί παιδί** του κόμβου (node->right) στο οποίο και εισάγεται η τιμή.
  Η μέθοδος δηλαδή καλείται έως ότου i) ο κόμβος που έχει ως όρισμα είναι κενός, όπου σε αυτή την περίπτωση δημιουργείται νέος κόμβος με τη νέα λέξη, ή ii) η λέξη που αντιστοιχεί στον κόμβο-όρισμα της μεθόδου είναι ίση με τη λέξη που εισάγεται, όπου σε αυτή την περίπτωση η μεταβλητή *count* του κόμβου αυξάνεται κατά ένα και τερματίζει.

- **NodeBST* deleteWord**: Η συγκεκριμένη μέθοδος αναπαριστά την **διαγραφή** μιας λέξης, ή την μείωσή της κατά 1 εάν υπάρχει πάνω από 1 φορά, από το δυαδικό δένδρο αναζήτησης. Έχει ως ορίσματα i) τον κόμβο που θα εξεταστεί (την πρώτη φορά που καλείται, ως όρισμα μπαίνει η ρίζα του δένδρου) ii) την λέξη που θα διαγραφεί από το δένδρο. Η μέθοδος καλείται αναδρομικά μέχρι να φτάσει στον κόμβο που βρίσκεται η λέξη ελέγχοντας όπως η μέθοδος **insertWord** τη λεξικογραφική θέση της λέξης σε σχέση με τη λέξη του εκάστοτε κόμβου-όρισμα.
  Όταν η μέθοδος φτάσει στον ζητούμενο κόμβο τότε:
  i) Εάν η λέξη υπάρχει παραπάνω από 1 φορά τότε απλά μειώνεται η μεταβλητή *count* κατά μία μονάδα και τερματίζει.

ii) Εάν ο κόμβος δεν έχει ούτε αριστερό ούτε δεξί παιδί τότε απλά διαγράφεται και τερματίζει καθώς πρόκειται για κόμβο-κλαδί και δεν επηρεάζει το υπόλοιπο δένδρο.
iii) Εάν ο κόμβος έχει μόνο δεξί παιδί τότε διαγράφεται και αντικαθίσταται από τον δεξί κόμβο-παιδί.
iv) Εάν ο κόμβος έχει μόνο αριστερό παιδί τότε διαγράφεται και αντικαθίσταται από τον αριστερό κόμβο-παιδί.
v) Εάν ο κόμβος έχει και δεξί και αριστερό παιδί τότε το πρόγραμμα εξετάζει τον κόμβο του δεξιού παιδιού. Εάν έχει αριστερό παιδί τότε εξετάζει τον κόμβο του αριστερού παιδιού μέχρις ότου φτάσει σε έναν κόμβο-παιδί που δεν έχει αριστερό παιδί. Για παράδειγμα:



Έστω ότι καλείται να διαγραφεί ο κόμβος 50. Τότε η μέθοδος θα ξεκινήσει από το δεξί κόμβο-παιδί 70. Θα ελέγξει αν υπάρχει αριστερό παιδί. Στο συγκεκριμένο παράδειγμα υπάρχει το 65. Οπότε συνεχίζει από το κόμβο 65 και ελέγχει εάν υπάρχει αριστερό παιδί. Υπάρχει το παιδί 62. Οπότε και συνεχίζει τον έλεγχο από τον κόμβο 62. Βρίσκει ότι δεν υπάρχει άλλο αριστερό παιδί οπότε το 62 αντικαθιστά το 50 και έπειτα τερματίζει. Το δυαδικό δένδρο γίνεται:



- **int searchWord:** Όπως και οι προηγούμενες μέθοδοι, η **searchWord** καλείται αναδρομικά μέχρι να φτάσει στην λέξη που δόθηκε ως παράμετρος. Εάν καταλήξει σε κενό κόμβο σημαίνει ότι η λέξη δεν υπάρχει στο δένδρο οπότε επιστρέφει 0. Σε διαφορετική περίπτωση επιστρέφει τον συνολικό αριθμό ύπαρξης της λέξης (μεταβλητή *count*).

- **void inOrder:** Διασχίζει το δυαδικό δένδρο περνώντας πρώτα από το αριστερό υποδένδρο, έπειτα από τον κόμβο και τέλος από το δεξί υποδένδρο. Το αποτέλεσμα είναι η διάσχιση των κόμβων του δένδρου κατά αύξουσα σειρά ως προς το περιεχόμενο τους.

- **void preOrder:** Διασχίζει το δυαδικό δένδρο περνώντας πρώτα από τον κόμβο, έπειτα από το αριστερό υποδένδρο και τέλος από το δεξί υποδένδρο

- **void postOrder**: Διασχίζει το δυαδικό δένδρο περνώντας πρώτα από το αριστερό υποδένδρο, έπειτα από το δεξί και τέλος από τον κόμβο

AVL Tree

Η δομή **AVL** αποτελείται από τις κλάσεις:

α) **NodeAVL:**

Η κλάση **NodeAVL** αναπαριστά τους κόμβους του δυαδικού δένδρου. Γι' αυτό και χρησιμοποιούνται οι μεταβλητές:

- **string word**, που αναπαριστά την λέξη του συγκεκριμένου κόμβου
- **int count**, η οποία μετράει τις φορές που εισήλθε στο δένδρο η λέξη
- **NodeAVL* left**, που αναπαριστά το αριστερό παιδί του κόμβου
- **NodeAVL* right**, που αναπαριστά το δεξί παιδί του κόμβου
- **NodeAVL* height**, που αναπαριστά το ύψος του κόμβου

β) **AVL**:

Η κλάση **AVL** αποτελείται από έναν κενό κατασκευαστή, 6 μεθόδους, μία για κάθε λειτουργία του δένδρου που ζητήθηκε από την εκφώνηση (*εισαγωγή, διαγραφή, αναζήτηση, inorder, preorder, postorder*) και άλλες 5 που βοηθούν στην επεξεργασία των κόμβων του AVL δένδρου *(setBalance, setHeight, Height, leftRotate, rightRotate*). Αναλυτικότερα:

- **int setBalance:** Επιστρέφει το balance του κόμβου που έχει ως όρισμα, το οποίο ισούται με το άθροισμα του ύψους του αριστερού και του δεξιού παιδιού του. Εάν ο κόμβος δεν υπάρχει, επιστρέφει 0.

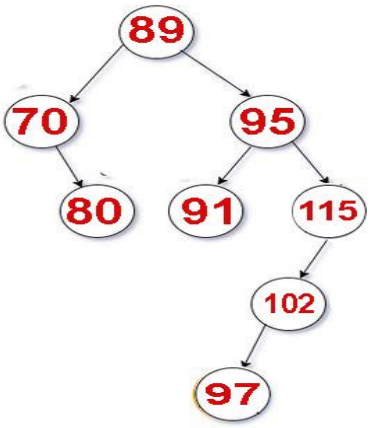
- **int height:** Επιστρέφει το ύψος του κόμβου που έχει ως όρισμα. Εάν ο κόμβος δεν υπάρχει επιστρέφει 0.

- **int setHeight:** Υπολογίζει το ύψος ενός κόμβου. Για τον υπολογισμό του ύψος ενός κόμβου χρειάζεται το ύψος του αριστερού και του δεξιού παιδιού. Βρίσκει ποιο από τα 2 είναι μεγαλύτερο και το επιστρέφει αυξημένο κατά 1.

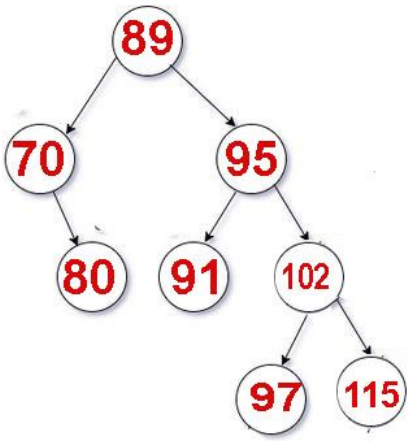
- **NodeAVL* leftRotate:** Η μέθοδος έχει ως όρισμα τον κόμβο και υλοποιεί την αριστερή περιστροφή του. Αυτό γίνεται ως εξής: ο κόμβος μετατρέπεται σε αριστερό παιδί του δεξιού παιδιού του. Οπότε το δεξί παιδί του κόμβου παίρνει την θέση του έχοντας ως αριστερό παιδί τον κόμβο-γονέα του, ενώ το δεξί του παιδί παραμένει ως έχει.

- **NodeAVL* rightRotate:** Η μέθοδος έχει ως όρισμα τον κόμβο και υλοποιεί την δεξιά περιστροφή του. Αυτό γίνεται ως εξής: ο κόμβος μετατρέπεται σε δεξί παιδί του αριστερού παιδιού του. Οπότε το αριστερό παιδί του κόμβου παίρνει την θέση του έχοντας ως δεξί παιδί τον κόμβο-γονέα του, ενώ το αριστερό του παιδί παραμένει ως έχει.

- **NodeAVL* insertWord:** Η εισαγωγή νέας λέξης στο AVL δένδρο ακολουθεί αρχικά τα βήματα της αντίστοιχης μεθόδου της **BinarySearchTree** κλάσης, αναζητώντας τον κατάλληλο κόμβο σύμφωνα με την λεξικογραφική θέση της λέξεις σε σχέση με τις υπόλοιπες του δένδρου. Το AVL δένδρο όμως είναι ισορροπημένο που σημαίνει ότι η διαφορά των υψών του κάθε κόμβου δεν πρέπει να είναι μεγαλύτερη από 1. Οπότε μετά την εισαγωγή του νέου κόμβου, η μέθοδος ελέγχει εάν χάλασε η ισορροπία του δένδρου, ελέγχοντας το balance των κόμβων που προσπελάστηκαν από το τέλος προς την αρχή. Ο κόμβος που εντοπίζεται πρώτος με τιμή balance διάφορη του -1, 0 ή 1 είναι ο κρίσιμος κόμβος και διακρίνονται οι εξής περιπτώσεις:
  i) Η τιμή balance του κρίσιμου κόμβου είναι μεγαλύτερη από 1 που σημαίνει ότι το αριστερό παιδί του έχει μεγαλύτερο ύψος από το δεξί, άρα και ο νέος κόμβος που δημιουργήθηκε, της καινούριας λέξης, είναι παιδί του αριστερού παιδιού του κρίσιμου κόμβου. Σε περίπτωση που είναι αριστερό παιδί του, τότε καλείται η συνάρτηση **rightRotate** ώστε να γίνει δεξιά περιστροφή ενώ σε περίπτωση που είναι δεξί παιδί τότε χρειάζεται να γίνει πρώτα **leftRotate** και μετά **rightRotate**.
  ii) Η τιμή balance του κρίσιμου κόμβου είναι μικρότερη από 1 που σημαίνει ότι το δεξί παιδί του έχει μεγαλύτερο ύψος από το αριστερό, όπως και πριν, θα πρέπει να επανέλθει η ισορροπία του δένδρου. Εάν ο νέος κόμβος είναι δεξί παιδί του δεξιού παιδιού του κρίσιμου κόμβου τότε καλείται η συνάρτηση **leftRotate** ώστε να γίνει αριστερή περιστροφή ενώ εάν είναι αριστερό παιδί τότε χρειάζεται να γίνει πρώτα **rightRotate** και έπειτα **leftRotate**. Για παράδειγμα:



Έστω ότι εισάγεται ο κόμβος με τον αριθμό 97. Η ισορροπία έχει χαλάσει καθώς η τιμή balance του 115 θα είναι ίση με 2. Άρα κρίσιμος κόμβος είναι ο 115. Ο κόμβος είναι αριστερό παιδί του 102 άρα θα χρειαστεί να γίνει δεξιά περιστροφή.



- **NodeAVL *deleteWord:** Η μέθοδος αρχικά ακολουθεί τα βήματα της αντίστοιχης μεθόδου της **BinarySearchTree** κλάσης. Βρίσκει δηλαδή τη θέση του κόμβου μέσα στο δένδρο, τον διαγράφει, και αλλάζει τη θέση των κόμβων εάν χρειάζεται. Μετά τη διαγραφή όμως ελέγχεται εάν έχει χαλάσει η ισορροπία του δένδρου. Έτσι ακολουθώντας από το τέλος προς την αρχή τους κόμβους που διασχίστηκαν μέχρι να φτάσει η μέθοδος στον κόμβο που διαγράφηκε ελέγχει την τιμή balance. Ο κόμβος που θα συναντήσει πρώτος που έχει balance μεγαλύτερο από 1 ή μικρότερο από -1 είναι ο κρίσιμος κόμβος. Διακρίνονται οι εξής περιπτώσεις:
  i) Εάν το balance του κρίσιμου κόμβου είναι μεγαλύτερο από 1 τότε το αριστερό παιδί του έχει μεγαλύτερο ύψος από το δεξί. Έπειτα ελέγχεται το balance του αριστερού παιδιού. Εάν είναι μεγαλύτερο ή ίσο του 0, δηλαδή εάν έχει δύο κόμβους-παιδιά ή μόνο αριστερό παιδί τότε καλείται η μέθοδος **rightRotate** για να γίνει δεξιά περιστροφή. Εάν το balance είναι μικρότερο

του 0, δηλαδή το αριστερό παιδί έχει μόνο 1 δεξί παιδί τότε γίνεται πρώτα αριστερή περιστροφή και έπειτα δεξιά.
ii) Εάν το balance του κρίσιμου κόμβου είναι μικρότερο από 1 τότε το δεξί παιδί του έχει μεγαλύτερο ύψος από το αριστερό. Έπειτα ελέγχεται το balance του δεξιού παιδιού. Εάν είναι μικρότερο ή ίσο του 0 τότε γίνεται αριστερή περιστροφή ενώ εάν είναι μεγαλύτερο του 0 τότε γίνεται πρώτα δεξιά περιστροφή και έπειτα αριστερή.

- **int searchWord:** ίδια με *BinarySearchTree*.
- **void inOrder:** ίδια με *BinarySearchTree*.
- **void preOrder:** ίδια με *BinarySearchTree*.
- **void postOrder:** ίδια με *BinarySearchTree*.

Hash Table

Η δομή **Hash Table** αποτελείται από τις κλάσεις:

α) **NodeHash:**

Η κλάση **NodeHash** αναπαριστά τις λέξεις μέσα στο Hash Table. Χρησιμοποιήθηκε κυρίως για τη διαχείριση εισαγωγής μιας ήδη υπάρχουσας λέξης Αποτελείται από έναν κατασκευαστή που παίρνει ως όρισμα την νέα λέξη, και από 2 μεταβλητές:

- **string word**, που αναπαριστά την λέξη του συγκεκριμένου κελιού
- **int count**, η οποία μετράει τις φορές που εισήλθε στο Hash Table η λέξη

β) **HashTable**:

Η κλάση **HashTable** περιέχει 3 private μεταβλητές. Έναν πίνακα words τύπου *NodeHash*, μια int μεταβλητή *size* που προσδιορίζει το αρχικό μέγεθος του πίνακα και μια *int* μεταβλητή **position** που δείχνει σε κάποιο κελί του πίνακα words. Επίσης περιέχει έναν κατασκευαστή ο οποίος δημιουργεί δυναμικά τον πίνακα *words* και 4 μέθοδοι, 2 που χρησιμοποιούνται για την υλοποίηση των βασικών λειτουργιών του πίνακα κατακερματισμού (*εισαγωγή, αναζήτηση*) και άλλες 2 που βοηθούν στην επεξεργασία των κελιών του Hash Table. Πιο αναλυτικά:

- **int findHash**: η μέθοδος αυτή καθορίζει τη θέση στην οποία θα εισαχθεί η νέα λέξη. Χρησιμοποιήθηκε η μέθοδος της Java String η οποία λειτουργεί ως εξής: Έστω ότι εισάγεται η λέξη “test”. Ορίζεται η μεταβλητή hash με αρχική τιμή 0 ισχύει: **hash = hash*37 + word[i]** για i από 0 έως 3 (δηλαδή 4 επαναλήψεις, όσα και τα γράμματα τις λέξης test). Έπειτα η μεταβλητή hash ισοδυναμεί με το υπόλοιπο της διαίρεσης με τον αριθμό **15679** (hash mod 15679). Ο αριθμός 15679 είναι μια τυχαία εκτίμηση του συνόλου των λέξεων (χωρίς να συμπεριληφθούν λέξεις που εμφανίζονται πάνω από μία

φορά) καθώς δεν γνωρίζουμε το τελικό μέγεθος του πίνακα κατακερματισμού. Επιλέχθηκε αυτός ο αριθμός γιατί είναι *πρώτος* (*prime*) και αυτό έχει ως αποτέλεσμα τη μείωση των *collisions*.

- **void expandTable**: Καθώς δεν γνωρίζουμε το τελικό μέγεθος του πίνακα, χρειάζεται αυξάνεται το μέγεθος του κάθε φορά που μια λέξη πρόκειται να εισαχθεί σε θέση μεγαλύτερη από το μέγεθος του πίνακα.

- **void insertWord**: η μέθοδος *insertWord* υλοποιεί την εισαγωγή νέας λέξης στον πίνακα κατακερματισμού. Ορίζεται η θέση position που θα εισαχθεί από τη συνάρτηση *findHash*. Ξεκινώντας από την θέση position ελέγχεται εάν υπάρχει άλλη λέξη ήδη σε αυτή τη θέση. Σε περίπτωση που η ίδια λέξη έχει εισαχθεί ήδη στον πίνακα τότε απλά αυξάνεται η μεταβλητή count κατά 1 και έπειτα τερματίζει. Ένα η θέση position είναι πιασμένη αλλά όχι από την ίδια την λέξη τότε η θέση αυξάνεται κατά 37*(μέγεθος λέξης) και ελέγχεται ξανά, έως ότου φτάσει σε μια θέση του πίνακα words που είναι κενή.

- **int searchWord**: Τέλος η μέθοδος *searchWord* αναζητά την λέξη που της δόθηκε ως όρισμα ξεκινώντας από τη θέση position (η οποία προκύπτει από τη συνάρτηση *findHash*) και καταλήγοντας στο τέλος του πίνακα κατακερματισμού με βήμα 37*(μέγεθος λέξης).

main

Πέρα από την main() υπάρχουν άλλες 2 συναρτήσεις. Μία για την ανάγνωση των λέξεων από ένα text αρχείο και μία για την εμφάνιση του αριθμού επανάληψης λέξεων. Πιο συγκεκριμένα:

- **bool readWords:** Η συνάρτηση χρησιμοποιείται για την ανάγνωση λέξεων μέσα από το αρχείο input.txt που βρίσκεται στον φάκελο Project του προγράμματος. Σε περίπτωση που το άνοιγμα του αρχείο αποτύχει η συνάρτηση τερματίζει και επιστρέφει **false**. Η συνάρτηση διαβάζει το αρχείο γραμμή προς γραμμή. Κάθε γραμμή αποθηκεύεται στη μεταβλητή *line*. Έπειτα ελέγχονται οι χαρακτήρες της line ένας προς ένας. Κάθε χαρακτήρας αποθηκεύεται στο τέλος της μεταβλητής *word*. Κάθε κεφαλαίος χαρακτήρας μετατρέπεται σε μικρός. Εάν βρεθεί χαρακτήρας που δεν αντιστοιχεί σε γράμμα (*π.χ. τελεία, κώμα, κενό*) αυτό σημαίνει ότι έχουμε αλλαγή λέξης οπότε οι χαρακτήρες που είχαν αποθηκευτεί μέχρι τότε στη μεταβλητή word αποτελούν μία λέξη. Γίνεται εισαγωγή της λέξης στις δομές **BinarySearchTree**, **AVL Tree** και **HashTable**. Επίσης όταν φτάνει στον τελευταίο χαρακτήρα της γραμμής, εάν αυτός είναι γράμμα τότε εισάγεται η λέξη *word* στις 3 δομές καθώς έχουμε αλλαγή γραμμής. Μετά την εισαγωγή οι χαρακτήρες της μεταβλητής *word* σβήνονται ώστε να εισαχθούν ξανά οι

χαρακτήρες της επόμενης λέξης. Παράλληλα με την εισαγωγή των λέξεων στις δομές, **Q** τυχαίες λέξεις από αυτές αποθηκεύονται στον πίνακα *random*.

- **void searchRandomWord:** Αναζητείται κάθε λέξη του πίνακα random και από τις 3 δομές και καταγράφεται ο χρόνος που χρειάστηκε μέχρι να βρεθούν. Για την καταγραφή του χρόνου χρησιμοποιείται το **steady_clock** της βιβλιοθήκης **chrono**.

Συμπέρασμα

Κατά μέσο όρο η αναζήτηση μιας λέξης επιτυγχάνεται γρηγορότερα με τη χρήση του **Hash Table**. Έπειτα ακολουθεί το **AVL Tree** και τέλος το **Binary Search Tree**. Τα αποτελέσματα δεν είναι πάντα τα ίδια και εξαρτώνται κυρίως από τις λέξεις που αποθηκεύτηκαν στον πίνακα random. Εάν υπάρχουν λέξεις που η εισαγωγή τους έγινε στην αρχή τότε η αναζήτηση τους στο **Binary Search Tree** είναι γρηγορότερη από τις άλλες 2 δομές