ΕΡΓΑΣΙΑ: ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Ονοματεπώνυμο	Πέρρος Κωνσταντίνος	Χρήστος Λυμούσης
ΑΕΜ	3713	3753
Ακαδημαϊκό email	knperros@csd.auth.gr	clymousis@csd.auth.gr
Έτος	1ο	1ο

Εισαγωγή

Για διευκόλυνση ανάγνωσης του κώδικα και των σχολίων, δίνονται οι παρακάτω υπενθυμίσεις:

- Οποιοδήποτε κομμάτι του κώδικα βρίσκεται μέσα σε '//', δείχνει είτε σχόλια ως μέρος του κώδικα στο οποίο βρισκόμαστε (π.χ. //LIBRARIES, το οποίο θυμίζει πως παρακάτω αναφέρονται όλες οι βιβλιοθήκες που χρησιμοποιούνται), είτε αχρησιμοποίητες γραμμές κώδικα που αγνοούνται.
- Οποιοδήποτε κομμάτι του κώδικα βρίσκεται μέσα σε '///', επεξηγεί την λειτουργία ευρέων κομματιών κώδικα, άλλα και μεμονωμένων εντολών. Τα κομμάτια κώδικα που βρίσκονται ανάμεσα σε γραμμές σχολίων '[OBSOLETE]', δείχνουν επίσης αχρησιμοποίητο κώδικα, ο οποίος αγνοείται.
- Όλες οι γραμμές κώδικα που βρίσκονται ανάμεσα σε '/*' και '*/' δείχνουν αχρησιμοποίητο κώδικα ή κώδικα για δοκιμές, συνεπώς αγνοούνται.
- Το πρόγραμμα γράφτηκε χρησιμοποιώντας codeblocks, στο οποίο όλα τα αρχεία αναλύονται ομοιόμορφα για οπτική διευκόλυνση.

main.cpp

Το βασικό αρχείο στο οποίο καλείται η **main()**, η οποία καλεί και δημιουργεί όλες τις δομές δεδομένων, καθώς και διαβάζει και γράφει αντίστοιχα αρχεία.

Αρχικά, αναγράφονται όλες οι βιβλιοθήκες που θα χρησιμοποιηθούν, καθώς και όλες οι κλάσεις που καλούνται. Η βιβλιοθήκη <fstream> χρησιμοποιείται για ανάγνωση και δημιουργία αρχείων κειμένου. Η βιβλιοθήκη <string> χρησιμοποιείται ώστε να επωφεληθούμε των πλεονεκτημάτων μεταβλητών **string** αντί για **char ***. Η <iostream> για έξοδο δεδομένων στην κονσόλα άλλα και για επιπρόσθετους μικρότερους λόγους. Η <bits/stdc++.h> για διάφορες χρήσεις όπως στην **chrono** και στην **isalpha()**. Τέλος, η <time.h> για την χρήση της ώρας του υπολογιστή, για την **srand()** και για διάφορες χρήσεις.

Έπειτα, μέσα στην **main()**, δημιουργούνται όλες οι μεταβλητές που θα χρησιμοποιηθούν αργότερα (π.χ. κλάσεις, χρόνοι, αριθμός λέξεων, ονόματα αρχείων κλπ.) και γίνεται ανάγνωση του αρχείο «**small-file.txt**». Υπάρχει η δυνατότητα να αλλάξει το όνομα του αρχείου που θα

διαβαστεί είτε μέσα απ' την **main.cpp**, είτε χρησιμοποιώντας την κονσόλα για είσοδο ονόματος αρχείου, άλλα καθώς για την εργασία αυτή δεν διαβάζεται τίποτα από το πληκτρολόγιο, το όνομα μπορεί να αλλάξει μόνο με τον πρώτο τρόπο.

Το αρχείο κειμένου διαβάζεται επαναληπτικά. Αρχικά, κάθε γραμμή του κειμένου αποθηκεύεται προσωρινά. Πάνω σε αυτήν την γραμμή, διαβάζεται επαναληπτικά κάθε στοιχείο, ελέγχοντας αν είναι χαρακτήρας (μόνο γράμματα στο αγγλικό αλφάβητο γίνονται αποδεκτά). Μόλις το πρόγραμμα συναντήσει στοιχείο που δεν είναι χαρακτήρας, η λέξη αυτή καταγράφεται. Οι έλεγχοι αυτοί γίνονται χρησιμοποιώντας την συνάρτηση **isalpha().** Μόλις ολοκληρωθεί η ανάγνωση της γραμμής κειμένου, η ίδια διαδικασία γίνεται για την επόμενη γραμμή κειμένου, ώσπου να τελειώσει το αρχείο.

Στην πρώτη επανάληψη για την ανάγνωση του αρχείου, το μόνο που πραγματοποιείται είναι η καταμέτρηση των λέξεων που υπάρχουν σε αυτό. Είναι σημαντικό αυτό, γιατί αλλιώς δεν μπορούμε να δημιουργήσουμε ένα hash table ή τυχαίο σύνολο Q για την τυχαία αναζήτηση.

Στην δεύτερη επανάληψη, κάθε λέξη που καταγράφεται αποθηκεύεται προσωρινά και εισάγεται σε όλες τις δομές (Binary Search Tree, AVL Tree, Hash Table), εισάγοντας έτσι όλες τις λέξεις του κειμένου στις δομές.

Στην τρίτη επανάληψη, γίνεται τυχαία αναζήτηση λέξεων στις δομές. Αρχικά, επιλέγεται ένα τυχαίο σύνολο λέξεων απ' το κείμενο χρησιμοποιώντας την **rand()** και την ώρα του υπολογιστή. Έπειτα, κάθε λέξη του κειμένου που καταγράφεται έχει 50% πιθανότητα να αναζητηθεί, μέχρι να αναζητηθούν όλες οι λέξεις απ' το σύνολο λέξεων που επιλέχτηκε. Εναλλακτικά, το αρχείο διαβάζεται απ' την αρχή. Κάθε αναζήτηση που γίνεται σε κάθε δομή χρειάζεται έναν χρονικό διάστημα για να εκτελεστεί, ο οποίος καταγράφεται και αποθηκεύεται. Επίσης, κάθε λέξη που αναζητείται σε κάθε δομή καταγράφεται σε ένα αρχείο «**output.txt**», καθώς και ο αριθμός ύπαρξής του σε κάθε δομή.

Δυστυχώς, η **rand()** δίνει τυχαίους αριθμούς μόνο έως ένα συγκεκριμένο όριο, οπότε είναι αδύνατο να αναζητηθεί αρκετά μεγάλο μέρος κειμένου ή να αποκαλεστεί πραγματικά τυχαία η αναζήτηση αυτή. Δεν βρήκαμε όμως άλλους τρόπους για πιο αποτελεσματική τυχαία αναζήτηση, καθώς δεν είχαμε περαιτέρω γνώσεις στο ζήτημα.

Στο τέλος, αποθηκεύεται στο αρχείο «**output.txt**» ο συνολικός χρόνος αναζήτησης κάθε δομής και το πρόγραμμα τερματίζει. Σε αυτό το σημείο, πριν τον τερματισμό, μπορεί επίσης να δοκιμαστεί ελεύθερα και η διαγραφή στις δομές.

Binary_Search_Tree.h

Για αυτό το αρχείο που αναπαριστά το δυαδικό δέντρο δεν υπάρχουν πολλά να ειπωθούν καθώς περιλαμβάνει μόνο τα headers των συναρτήσεων για την κλάση.

Οι βιβλιοθήκες που περιλαμβάνονται είναι η <string.h> και η <iostream.h> για προφανείς λόγους.

Δημιουργείται επίσης και μια βασική δομή **struct node** πάνω στην οποία χτίζεται κάθε φύλλο του δυαδικού δέντρου, περιέχοντας πληροφορίες για την τιμή της λέξης, τον αριθμό ύπαρξής της, την θέση στην μνήμη των παιδιών της και τέλος το ύψος της, το οποίο χρησιμοποιείται μόνο στο AVL δέντρο.

Στις δημόσιες παραμέτρους βρίσκονται ο κατασκευαστής της κλάσης, οι συναρτήσεις Εισαγωγής, Αναζήτησης και Διαγραφής (**Insert()**, **Search()** και **Delete()** αντίστοιχα), οι συναρτήσεις **preorder()**, **inorder()** και **postorder()**, καθώς και οι βοηθητικές συναρτήσεις **debugInfo()** και **printDebug()** για δοκιμές, και τέλος η **Getroot()** που επιστρέφει την ρίζα του δέντρου.

Στις προστατευμένες παραμέτρους βρίσκονται οι γενικές εφαρμογές όλων των παραπάνω συναρτήσεων. Βρίσκονται στις προστατευμένες παραμέτρους, διότι θέλουμε να μην χρησιμοποιούνται από ξένες κλάσεις, άλλα ταυτόχρονα να έχουν πρόσβαση κληρονομούσες κλάσεις, όπως η **AVL_Tree.h**. Επίσης, διάφορες βοηθητικές συναρτήσεις που χρησιμοποιούνται απ' τις κύριες παραπάνω συναρτήσεις βρίσκονται εδώ και είναι η **node_replacement()**, με βασικό σκοπό την εύκολη αντικατάσταση κόμβων που πρόκειται να διαγραφούν, **string_comparison()**, που δεν χρησιμοποιείται πλέον και η **children_comparison()**, που χρησιμοποιείται στην διαγραφή για την εύρεση των αριθμών παιδιών ενός κόμβου. Τέλος, η συνάρτηση **parent_search()** βρίσκει τον γονέα του κόμβου στον οποίο υπάρχει η λέξη που δίνεται.

Στις ιδιωτικές παραμέτρους βρίσκεται μόνο η **root**, δηλαδή η ρίζα του δέντρου.

Binary_Search_Tree.cpp

Η ανάλυση κάθε συνάρτησης του header γίνεται εδώ.

Για αρχή μηδενίζεται η ρίζα του δέντρου στον κατασκευαστή (**nullptr**). Έπειτα υλοποιούνται οι συναρτήσεις **preorder()**, **inorder()** και **postorder()**, που είναι τύπου **node ***, κάνοντας αρχικά κλήση των συναρτήσεων στην **main()**. Ύστερα καλούνται αναδρομικά μέσα στην κλάση με ανάλογο τρόπο η κάθε μία.

Insert

Η αναζήτηση γίνεται με παρόμοιο τρόπο, καλώντας στην **main()** την δημόσια συνάρτηση, και έπειτα καλώντας μέσα στην κλάση την προστατευμένη **Insert()**.

Κάθε φορά που δίνεται μία λέξη, η συνάρτηση ελέγχει αν ο τωρινός κόμβος που βρισκόμαστε είναι κατειλημμένος. Εάν δεν είναι, η λέξη αποθηκεύεται και δημιουργούνται τα παιδιά του. Εάν είναι κατειλημμένος ο κόμβος, η τιμή της λέξης συγκρίνεται με την τιμή του τωρινού κόμβου.

Αν η τιμή είναι μεγαλύτερη, η ίδια παραπάνω διαδικασία επαναλαμβάνεται για το δεξί παιδί του κόμβου. Διαφορετικά, η διαδικασία επαναλαμβάνεται για το αριστερό παιδί του. Αν η τιμή της λέξης είναι ίδια, ο αριθμός ύπαρξης της λέξης σε αυτόν τον κόμβο αυξάνεται κατά ένα. Η διαδικασία αυτή γίνεται αναδρομικά. Η συνάρτηση είναι τύπου **node ***.

Search

Παρόμοια με την Εισαγωγή λειτουργεί και η Αναζήτηση. Καλείται η συνάρτηση στην **main()** και έπειτα καλείται η προστατευμένη **Search()** μέσα στην κλάση.

Κάθε λέξη που δίνεται ελέγχεται αναδρομικά στους κόμβους του δέντρου. Αν ο τωρινός κόμβος δεν έχει την ίδια τιμή με την δοσμένη λέξη, τότε:

Μετακινούμαστε στο δεξί του παιδί, αν η τιμή της δοσμένης λέξης είναι μεγαλύτερη, ή στο αριστερό του παιδί, αν η τιμή τής λέξης είναι μικρότερη του κόμβου.

Αν η τιμή του κόμβου είναι ίδια με την τιμή της δοσμένης λέξης, τότε η λέξη υπάρχει και έχει βρεθεί. Εναλλακτικά η λέξη δεν υπάρχει και επιστρέφεται αντίστοιχο μήνυμα. Η συνάρτηση είναι τύπου **node *** και επιστρέφει τον αριθμό ύπαρξης της λέξης στο δέντρο.

Delete

Σε αντίθεση με όλες τις παραπάνω συναρτήσεις, επιλέξαμε να υλοποιήσουμε αλλιώς την **Delete()**. Παρόλο που η συνάρτηση καλείται αρχικά στην **main()** και έπειτα επαναληπτικά στην κλάση προστατευμένα, η διαδικασία ***δεν*** γίνεται αναδρομικά και η συνάρτηση είναι τύπου **bool**.

Όταν δίνεται μία λέξη για διαγραφή, αρχικά γίνεται αναζήτηση στην δομή χρησιμοποιώντας την **Search()**. Εάν δεν υπάρχει η λέξη τότε επιστρέφεται **false.**

Εάν η λέξη υπάρχει, τότε καλείται η συνάρτηση **parent_search()** ώστε να βρεθεί ο γονέας του κόμβου που πρόκειται να διαγραφεί. Έπειτα καλείται η συνάρτηση **children_comparison()**, η οποία επιστρέφει έναν αριθμό από 0 έως 3 και χρησιμοποιείται στην ακριβώς παρακάτω **switch case**, που αντιστοιχεί στην ύπαρξη παιδιών του κόμβου που πρόκειται να διαγραφεί:

- Αν επιστραφεί 0, τότε ο κόμβος που πρόκειται να διαγραφεί δεν έχει παιδιά, οπότε απλώς διαγράφεται και ο γονέας πλέον δείχνει στο **nullptr**. Η αντικατάσταση του κόμβου γίνεται μέσω της **node_replacement()** που αντικαθιστά το αντίστοιχο παιδί του γονέα του κόμβου με το κενό.

- Αν επιστραφεί 1, τότε ο κόμβος έχει μόνο δεξί παιδί, οπότε ο γονέας του πλέον θα δείχνει μόνο στο δεξί παιδί του κόμβου. Η αντικατάσταση του κόμβου γίνεται μέσω της **node_replacement()** που αντικαθιστά το αντίστοιχο παιδί του γονέα του κόμβου με το δεξί παιδί του κόμβου.

- Αν επιστραφεί 2, τότε ο κόμβος έχει μόνο αριστερό παιδί, οπότε ο γονέας του πλέον θα δείχνει μόνο στο αριστερό παιδί του κόμβου. Η αντικατάσταση του κόμβου γίνεται μέσω της **node_replacement()** που αντικαθιστά το αντίστοιχο παιδί του γονέα του κόμβου με το αριστερό παιδί του κόμβου.

- Αν επιστραφεί 3, τότε ο κόμβος έχει 2 παιδιά. Στην συγκεκριμένη περίπτωση, δημιουργείται μια μεταβλητή τύπου **node *** με όνομα **end_node**, η οποία θα πάρει την θέση του κόμβου αυτού. Η διαδικασία εύρεσης του **end_node** γίνεται βρίσκοντας το αριστερότερο παιδί του δεξιού παιδιού του κόμβου (δηλαδή αρχικά πάμε στο δεξί παιδί του κόμβου που πρόκειται να διαγραφεί και έπειτα επαναληπτικά σε κάθε αριστερό παιδί που υπάρχει μέχρι να βρεθούμε σε έναν κόμβο χωρίς αριστερό παιδί). Αφού βρεθεί ο **end_node**, βρίσκουμε τον γονέα του **end_node**. Αν ο γονέας αυτός είναι ίσος με τον κόμβο που θέλουμε να διαγράψουμε, τότε η διαδικασία είναι πιο εύκολη, μεταφέροντας απλά τα παιδιά τού κόμβου στον **end_node** και βάζοντας τον γονέα του κόμβου που πρόκειται να διαγραφεί να δείχνει στο **end_node** με την βοήθεια του **node_replacement()**. Διαφορετικά, κάνουμε την ίδια διαδικασία, αλλά επιπλέον αντικαθιστούμε το αριστερό παιδί του γονέα του **end_node** είτε με το **nullptr**, είτε με το δεξί παιδί του **end_node** αν υπάρχει.

Στο τέλος ο κόμβος στον οποίο βρέθηκε αρχικά η λέξη που δόθηκε διαγράφεται. Εάν η διαδικασία ολοκληρώθηκε χωρίς πρόβλημα και η λέξη διαγράφηκε, τότε επιστρέφεται **true**, αλλιώς επιστρέφεται **false.**

AVL_Tree.h

Το AVL δέντρο για διευκόλυνση κληρονομεί την Binary_Search_Tree.h, για αυτό και δεν συμπεριλαμβάνονται επιπλέον βιβλιοθήκες, **structs** και συναρτήσεις. Η μόνη διαφορά απ' το δυαδικό δέντρο που έχει αυτή η δομή είναι οι περιστροφές.

Στις δημόσιες παραμέτρους βρίσκεται ο κατασκευαστής της κλάσης, προερχόμενος απ' το δυαδικό δέντρο καθώς και οι **Insert()**, **Search()**, **Delete()**, **printDebug()**, **preorder()**, **inorder()** και **postorder()**, όλες εκ των οποίων προέρχονται απ' το δυαδικό δέντρο και έχουν τις ακριβώς ίδιες λειτουργίες, εκτός της **Insert()**.

Στις ιδιωτικές παραμέτρους βρίσκεται η ρίζα **root**, όπως και στο δυαδικό δέντρο, η αναδρομική συνάρτηση **Insert()** που λειτουργεί ελαφρώς διαφορετικά του δυαδικού δέντρου και οι περιστροφικές συναρτήσεις:

- **balance_factor_check()** τύπου **node ***, η οποία αναδρομικά ελέγχει όλα τα υποδέντρα της ρίζας και αποφασίζει αν θα κάνει περιστροφές και ποιες.
- **height_update()** τύπου **int**, η οποία αλλάζει το ύψος του κόμβου που δίνεται
- **delete_avl_rotation_check()** τύπου **node ***, η οποία ελέγχει ένα δέντρο αν χρειάζεται περιστροφές μετά από μια διαγραφή κόμβου
- **leftRotate()** τύπου **node ***, που εκτελεί μια αριστερή περιστροφή.
- **rightRotate()** τύπου **node ***, που εκτελεί μια δεξιά περιστροφή.

AVL_Tree.cpp

Η ανάλυση των συναρτήσεων του header γίνεται εδώ.

Ο κατασκευαστής, προερχόμενος απ' το δυαδικό δέντρο, θέτει την ρίζα σε **nullptr**. Έπειτα, όλες οι συναρτήσεις που προέρχονται απ' το δυαδικό δέντρο καλούνται μόνο δημόσια, καλώντας και εκτελώντας έπειτα τις εσωτερικές συναρτήσεις του δυαδικού δέντρου. Για αυτό οι δημόσιες παράμετροι του δυαδικού δέντρου που μοιράζεται το AVL είναι **virtual**.

Οι συναρτήσεις **preorder()**, **inorder()** και **postorder()**, καθώς και η **printDebug()** και η **Search()** καλούνται προερχόμενες απ' το δυαδικό δέντρο οπότε δεν υπάρχουν πολλά να ειπωθούν.

Insert

Αρχικά, όπως και στην **Binary_Search_Tree.cpp**, η **Insert()** καλείται πρώτα δημόσια στην **main()**, και έπειτα ιδιωτικά μέσα στην κλάση. Η διαδικασία γίνεται αναδρομικά και δεν έχει καμία διαφορά από το δυαδικό δέντρο. Όμως πριν τελειώσει η κάθε αναδρομή και επιστραφεί ο κόμβος πίσω, εκτελείται η συνάρτηση **balance_factor_check()** στον κόμβο που γίνεται η αναδρομή.

Η **balance_factor_check()** αποθηκεύει το σχετικό ύψος του κόμβου μέσω της **height_update()** και δημιουργεί το αντίστοιχο **balance_factor** του κόμβου. Εάν ο κόμβος όμως είναι κενός (**nullptr**) η διαδικασία δεν συνεχίζει περαιτέρω και τερματίζει. Το **balance_factor** του κόμβου καθορίζεται αφαιρώντας το σχετικό ύψος του δεξιού παιδιού του κόμβου από το

σχετικό ύψος του αριστερού παιδιού του κόμβου. Το σχετικό ύψος του κάθε παιδιού βρίσκεται με την βοήθεια του **height_update()** και το αποτέλεσμα του **balance_factor** μπορεί να είναι μεγαλύτερο, μικρότερο ή ίσο με το 0.

- Αν η τιμή του είναι 0, τότε η διαδικασία τερματίζει και επιστρέφεται πίσω ο κόμβος
- Αν η τιμή του είναι μεγαλύτερη του 1 τότε:

    a) Αν η τιμή της λέξης του κόμβου είναι μικρότερη της τιμής της λέξης του αριστερού παιδιού του κόμβου, τότε θα γίνει μια δεξιά περιστροφή στον κόμβο χρησιμοποιώντας την **rightRotate()**.

    b) Αν η τιμή της λέξης του κόμβου είναι μεγαλύτερη της τιμής της λέξης του αριστερού παιδιού του κόμβου, τότε θα γίνει μια αριστερή περιστροφή στο αριστερό παιδί του κόμβου εφόσον υπάρχει το δεξί παιδί του κόμβου. Αυτό θα γίνει με την βοήθεια της **leftRotate()**. Στην συνέχεια, θα γίνει μια δεξιά περιστροφή στον κόμβο χρησιμοποιώντας την **rightRotate()**.

- Αν η τιμή του είναι μικρότερη -1 τότε:

    a) Αν η τιμή της λέξης του κόμβου είναι μεγαλύτερη της τιμής της λέξης του δεξιού παιδιού του κόμβου, τότε θα γίνει μια αριστερή περιστροφή στον κόμβο χρησιμοποιώντας την **leftRotate()**.

    b) Αν η τιμή της λέξης του κόμβου είναι μικρότερη της τιμής της λέξης του δεξιού παιδιού του κόμβου, τότε θα γίνει μια δεξιά περιστροφή στο δεξί παιδί του κόμβου εφόσον υπάρχει το αριστερό παιδί του κόμβου. Αυτό θα γίνει με την βοήθεια της **rightRotate()**. Στην συνέχεια, θα γίνει μια αριστερή περιστροφή στον κόμβο χρησιμοποιώντας την **leftRotate()**.

Η **height_update()** που έχει προαναφερθεί αρκετά, ουσιαστικά εφόσον δεν είναι κενός ο κόμβος, παίρνει το ύψος το αριστερού παιδιού του κόμβου και το ύψος του δεξιού παιδιού του κόμβου, και αναθέτει την μεγαλύτερη τιμή στο ύψος του κόμβου + 1. Αν ο κόμβος είναι κενός, τότε επιστρέφει το 0 ως ύψος.

LeftRotate

Η **leftRotate()** δέχεται έναν αρχικό κόμβο **current_node** και δημιουργεί δύο καινούριες μεταβλητές τύπου **node ***, έναν προσωρινό κόμβο **temp_node** και έναν τελικό κόμβο **end_node**.

Αρχικά, αναθέτει στον **end_node** το *δεξί παιδί* τού **current_node**. Εάν αυτό είναι κενό, η διαδικασία τερματίζει και ο **current_node** επιστρέφεται απαράλλαχτος.

Αν δεν είναι κενό, τότε ανατίθεται στον **temp_node** το *αριστερό παιδί* τού **end_node**. Έπειτα πραγματοποιείται περιστροφή, στην οποία το *αριστερό παιδί* τού **end_node** αντικαθίσταται με τον **current_node**, και το *δεξί παιδί* του **current_node** αντικαθίσταται με τον **temp_node**.

Αφού τελειώσει αυτή η διαδικασία, τα σχετικά ύψη του **current_node** και του **end_node** ενημερώνονται. Τέλος, επιστρέφεται ο **end_node**.

RightRotate

Η **rightRotate()** δέχεται έναν αρχικό κόμβο **current_node** και δημιουργεί δύο καινούριες μεταβλητές τύπου **node ***, έναν προσωρινό κόμβο **temp_node** και έναν τελικό κόμβο **end_node**.

Αρχικά, αναθέτει στον **end_node** το *αριστερό παιδί* τού **current_node**. Εάν αυτό είναι κενό, η διαδικασία τερματίζει και ο **current_node** επιστρέφεται απαράλλαχτος.

Αν δεν είναι κενό, τότε ανατίθεται στον **temp_node** το *δεξί παιδί* τού **end_node**. Έπειτα πραγματοποιείται περιστροφή, στην οποία το *δεξί παιδί* τού **end_node** αντικαθίσταται με τον **current_node**, και το *αριστερό παιδί* του **current_node** αντικαθίσταται με τον **temp_node**.

Αφού τελειώσει αυτή η διαδικασία, τα σχετικά ύψη του **current_node** και του **end_node** ενημερώνονται. Τέλος, επιστρέφεται ο **end_node**.

Delete

Η **Delete()** στο AVL δέντρο κληρονομείται από το δυαδικό δέντρο. Όμως αφού δεν γίνεται αναδρομικά, δεν μπορεί να γίνει η ίδια διαδικασία όπως στην **Insert()**. Αντιθέτως, κάθε φορά που διαγράφεται μία λέξη, εάν διαγράφηκε επιτυχώς, εκτελείται η βοηθητική συνάρτηση **delete_avl_rotation_check()**.

Η συνάρτηση **delete_avl_rotation_check()** ουσιαστικά εκτελεί μία αναδρομή απ’ την ρίζα του δέντρου προς όλα τα υπόδεντρα και εκτελεί για κάθε κόμβο την συνάρτηση **balance_factor_check()**. Αυτό θεωρητικά είναι αρκετά αργό, καθώς για κάθε λέξη που διαγράφεται, ελέγχεται όλο το δέντρο από την αρχή, φτιάχνοντας το ένα λάθος που μπορεί να έχει προκαλέσει η διαγραφή. Καθώς όμως δεν εκτελείται αναδρομικά η διαγραφή, δεν σκεφτήκαμε καλύτερο τρόπο για να γίνει η εκτέλεσή της.

Τα αποτελέσματα για την κάθε διαγραφή δεν είναι πολύ αργά, κάνοντας περίπου τον ίδιο χρόνο που κάνει και η **Search()**, οπότε παραμείναμε στην τωρινή της υλοποίηση ως τελική.

Hash_Table.h

Το Hash Table κληρονομεί το δυαδικό δέντρο, μόνο ώστε να χρησιμοποιεί τις ίδιες βιβλιοθήκες.

Περιέχει ένα **struct cell** το οποίο αναπαριστά το κάθε κελί του πίνακα Hash. Το κάθε κελί περιέχει πληροφορίες για την λέξη που έχει αποθηκευμένη και για τον αριθμό ύπαρξής της.

Στις δημόσιες παραμέτρους περιέχεται ένας κατασκευαστής και ένας καταστροφέας. Επίσης βρίσκονται οι συναρτήσεις **Insert()** για την Εισαγωγή δεδομένων στον πίνακα και **Search()** για Αναζήτηση στον πίνακα. Η βοηθητική συνάρτηση **insert_from_BTS()** δεν χρησιμοποιείται.

Στις ιδιωτικές παραμέτρους βρίσκεται η μεταβλητή **table** τύπου **cell ***, που αναπαριστά ολόκληρο τον πίνακα Hash, καθώς και η μεταβλητή **table_size** που δείχνει το μέγεθος του πίνακα αυτού. Επίσης, εμπεριέχονται και οι βοηθητικές συναρτήσεις **Cell_Occupation_Check()** για έλεγχο του κάθε κελιού, η **hash_help()**, η οποία αγνοείται, και τέλος η **Hash_Conversion()** που χρησιμοποιείται ως ο κύριος αλγόριθμος *hashing* του πίνακα.

Hash_Table.cpp

Αρχικά, ο κατασκευαστής παίρνει ως όρισμα το μέγεθος του πίνακα, το διπλασιάζει και το καταχωρεί στην μεταβλητή **table_size** και δημιουργεί έναν αντίστοιχο πίνακα **table** τύπου **cell *** μεγέθους **table_size**. Ο καταστροφέας διαγράφει τα δεδομένα του **table** και μηδενίζει το **table_size**.

Η συνάρτηση **Hash_Conversion()**, που είναι ο βασικός αλγόριθμος hashing, παίρνει ως όρισμα μια λέξη και εκτελεί μαθηματικές πράξεις με βάση το μέγεθος του πίνακα και το μέγεθος και τιμή της λέξης αυτής. Η τιμή που δημιουργείται είναι το κλειδί της λέξης, επιστρέφεται από την συνάρτηση και αναπαριστά το αντίστοιχο κελί στον πίνακα **table**. Όσο περίπλοκος και να είναι ο αλγόριθμος αυτός όμως, πιθανότητα μία λέξη να έχει το ίδιο κλειδί με μία άλλη δεν είναι μηδέν, για αυτό χρησιμοποιείται στις υπόλοιπές συναρτήσεις πρώτα η **Cell_Occupation_Check()**.

Η συνάρτηση **Cell_Occupation_Check()** δέχεται μία λέξη και δημιουργεί μία μεταβλητή **cell** τύπου **int** που παίρνει ως τιμή το αποτέλεσμα της **Hash_Conversion()** με την δοσμένη λέξη. Έπειτα, αναζητείται το κελί **cell** στον πίνακα **table** (**table[cell]**) και ελέγχεται αν ήδη υπάρχει μία λέξη στο κελί αυτό. Αν υπάρχει, η τιμή του cell αυξάνεται κατά ένα ή θέτει το κελί στην τιμή 0, αν δεν υπάρχουν περαιτέρω κελιά, και η διαδικασία ελέγχου επαναλαμβάνεται, έως ότου βρεθεί ένα κενό κελί στον πίνακα. Η συνάρτηση τελικά επιστρέφει την μεταβλητή **cell** που αναπαριστά το κελί της δοσμένης λέξης.

Insert

Η συνάρτηση Εισαγωγής **Insert()** δέχεται ως όρισμα μία λέξη και βρίσκει το αντίστοιχο κελί που μπορεί να εισαχθεί στον πίνακα **table** μέσω της συνάρτησης **Cell_Occupation_Check()**. Αν η λέξη βρίσκεται ήδη στον πίνακα (**table[cell]**), τότε απλώς αυξάνεται ο αριθμός ύπαρξής της κατά ένα. Διαφορετικά, το κελί καταλαμβάνεται και αυξάνεται ο αριθμός ύπαρξης της λέξης κατά ένα.

Search

Η συνάρτηση Αναζήτησης δέχεται ως όρισμα μία λέξη και αναζητά στον πίνακα την λέξη, μέσω της βοήθειας της **Cell_Occupation_Check()**. Όταν η λέξη βρεθεί, τότε επιστρέφεται ο αριθμός ύπαρξης της στον πίνακα.

Επίλογος

Οι τελικοί χρόνοι για την κάθε δομή είναι αρκετά γρήγοροι. Χρησιμοποιώντας το αρχείο «**small-file.txt**» και αναζητώντας όλες τις λέξεις του κειμένου, ο συνολικός χρόνος εκτέλεσης είναι περίπου 9 δευτερόλεπτα. Η πιο γρήγορη δομή είναι η Hash Table, ακολουθούμενη από την Binary Search Tree και τελικά την AVL Tree, η οποία είναι πιο αργή εξαιτίας των περιστροφών.

Είμαστε σίγουροι πως το πρόγραμμα δεν είναι άκρως αποδοτικό, άλλα πιστεύουμε πως είναι τουλάχιστον επαρκές ως προς τους χρόνους εκτέλεσης και τους τρόπους λειτουργίας. Επίσης, πολλές αχρησιμοποίητες γραμμές κώδικα θα μπορούσαν να διαγραφούν, άλλα παραμένουν για περαιτέρω δοκιμαστικούς σκοπούς.

Ευχαριστούμε.