ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Ονοματεπώνυμο	ΑΕΜ
Άννα Τζανακοπούλου	3471
Αναστάσιος Παντζαρτζής	3216

Για την υλοποίηση των δομών:

- Πίνακας
- Ταξινομημένος Πίνακας
- Δυαδικό Δέντρο Αναζήτησης
- AVL
- Πίνακας Κατακερματισμού (με ανοιχτή διεύθυνση)

Δημιουργήθηκαν 7 αρχεία .cpp και 6 αρχεία .h τα οποία θα αναλυθούν παρακάτω.

Main.cpp:

Αρχικά, υλοποιήθηκε η συναρτήσεις:

- **lowerCase()** η οποία είναι η υπεύθυνη για τη μετατροπή των λέξεων σε μικρά γράμματα, ώστε να έχουν όλες οι λέξεις την ίδια μορφή και να αποφευχθούν τα διπλότυπα, πχ Test -> test = test.
- **changeTheSizeNow()** δημιουργήθηκε με σκοπό να αλλάζει το μέγεθος των δυναμικών πινάκων **uniqueWords**, **totalWords**, που περιέχουν τις μοναδικές και τις συνολικές λέξεις του αρχείου, αντίστοιχα.
- **stripTheSymbols()** είναι υπεύθυνη για την διαγραφή των συμβόλων από τις λέξεις, καθώς και των κενών, πχ @apple! -> apple.
- **Main():**
    - Δήλωση και αρχικοποίηση όλων των απαραίτητων μεταβλητών (counters, file name, arrays, …).
    - Δημιουργία αντικειμένων για κάθε δομή δεδομένων.
    - Άνοιγμα αρχείου **file.txt** με χρήση try-catch σε περίπτωση σφάλματος.

- Μέσα σε ένα while-loop για κάθε λέξη καλείται η συνάρτηση lowercase() και stripTheSymbols() ώστε να μετατραπεί στην κατάλληλη μορφή. Ακολούθως, ελέγχουμε εάν η λέξη (μία κάθε φορά) υπάρχει στο AVL. Αν όχι, προσθέτουμε τη λέξη στον πίνακα με τις μοναδικές λέξεις. Ακόμη, σε περίπτωση μη συμβατότητας των μεγεθών καλείται η **changeTheSizeNow()** και προστίθεται η λέξη σε έναν άλλο πίνακα. Τέλος, για κάθε δομή καλούνται οι αντίστοιχες συναρτήσεις εκχώρησης (insert…, πχ **insertSortedArray()**).
- Με χρήση της μεθόδου rand() επιλέγεται με τυχαίο τρόπο ένα σύνολο λέξεων από τον πίνακα που έχουν αποθηκευτεί οι μοναδικές λέξεις, το οποίο αποτελεί το ~13% των λέξεων του αρχείου.
- Τα αποτελέσματα που θα προκύψουν εκτυπώνονται σε ένα αρχείο txt (output.txt).
- Στη συνέχεια, για κάθε δομή δεδομένων υπολογίζεται ο χρόνος που χρειάστηκε για την αναζήτηση (end time - start time).
- Στο αρχείο εξόδου output.txt εκτυπώνεται ο απαιτούμενος χρόνος για κάθε δομή, ο συνολικός αριθμός των λέξεων που αναζητήθηκαν, καθώς και οι μοναδικές λέξεις με το πλήθος των εμφανίσεων τους στο κείμενο.

ARRAY (Array)

Η δομή του πίνακα αποτελείται από τις λειτουργίες εισαγωγής, διαγραφής και αναζήτησης. Αναλυτικότερα,

- **deleteArray():** δέχεται ως όρισμα τη λέξη που πρόκειται να διαγραφτεί. Για να γίνει αυτό αναζητεί μέσα στον πίνακα τη λέξη, αν τη βρει αλλάζει η τιμή μίας τύπου bool μεταβλητής σε true και αποθηκεύεται η θέση που βρέθηκε η λέξη. Ακόμη αν έχει βρεθεί η δοθείσα λέξη τότε μειώνεται η διάσταση του πίνακα κατά ένα και διαγράφεται το στοιχείο.
- **insertArray():** δέχεται ως όρισμα τη λέξη που πρόκειται να εισαχθεί στον πίνακα. Αν δεν υπάρχει επαρκής χώρος στον πίνακα διπλασιάζεται το μέγεθός του, αντιγράφονται τα στοιχεία από τον αρχικό στον νέο πίνακα και διαγράφεται ο αρχικός. Τέλος εισάγεται το στοιχείο στο τέλος του πίνακα.
- **theSearch():** δέχεται ως όρισμα τη λέξη που πρόκειται να αναζητηθεί. Με χρήση ενός for-loop αναζητά την λέξη. Αν βρεθεί, η κατάσταση της λογικής μεταβλητής isFound αλλάζει σε true.

SORTED ARRAY (SortedArray)

Η δομή του πίνακα αποτελείται από τις λειτουργίες εισαγωγής, διαγραφής και αναζήτησης. Αναλυτικότερα,

- **insertSortedArray():** Δέχεται ως όρισμα τη λέξη που πρόκειται να εισαχθεί και δύο τιμές **low**, **high**. Η συνάρτηση υλοποιεί τον αλγόριθμο της quicksort. Αν η μικρότερη τιμή που δέχεται είναι μικρότερη από τη μεγαλύτερη τότε καλεί τη συνάρτηση **partition** και καλεί αναδρομικά τον εαυτό της.
- **searchInSortedArray():** δέχεται ως όρισμα τη λέξη που πρόκειται να αναζητηθεί. Με χρήση ενός while-loop αναζητά την λέξη. Αν βρεθεί, η κατάσταση της λογικής μεταβλητής isFound αλλάζει σε true και κρατάει τη θέση στην οποία βρέθηκε η λέξη.
- **deleteInSortedArray():** δέχεται ως όρισμα τη λέξη που πρόκειται να διαγραφτεί. Βρίσκει τη θέση που έχει η λέξη στον πίνακα με την βοήθεια της **searchInSortedArray().** Ακόμη αν έχει βρεθεί η δοθείσα λέξη τότε μειώνεται η διάσταση του πίνακα κατά ένα και διαγράφεται το στοιχείο.
- **partition():** Ορίζει το τελευταίο στοιχείο ως pivot και το μετακινεί στην κατάλληλη θέση στον ταξινομημένο πίνακα. Αν το στοιχείο είναι μικρότερο από το στοιχείο pivot τότε πηγαίνει αριστερά από αυτό, αλλιώς δεξιά του pivot (χρήση της swap για τις αντιμεταθέσεις).
- **swap():** αντιμεταθέτει τα στοιχεία που έχουν οι διευθύνσεις που λαμβάνει ως ορίσματα.

Binary Search Tree (BinaryTree)

Η δομή του δυαδικού δέντρου αναζήτησης αποτελείται από τις λειτουργίες εισαγωγής, διαγραφής και αναζήτησης, καθώς και από τις προσπελάσεις **preorder**, **postorder** και **inorder**. Αναλυτικότερα,

- **InsertElement():** Δέχεται ως ορίσματα την λέξη που θα εισαχθεί καθώς και το node της ρίζας.
    - Αν η λέξη που έχει δοθεί ως όρισμα βρεθεί ήδη μέσα στην δομή τότε αυξάνει το πλήθος της συγκεκριμένης λέξης.
    - Αν το node που ελέγχεται είναι κενό (NULL) τότε δημιουργείται νέο αντικείμενο.
    - Αλλιώς καλείται η **InsertElement()** με αναδρομή (η εισαγωγή γίνεται με βάσει τη λεξικογραφική σειρά. Η διαδικασία σταματάει αν βρεθεί η λέξη ή κενό node.
- **DeleteElement():** Δέχεται ως ορίσματα την λέξη που θα εισαχθεί καθώς και το node της ρίζας.
    - Αν το node που ελέγχεται είναι κενό (NULL) τότε το επιστρέφει.
    - Αν η λέξη είναι μεγαλύτερη λεξικογραφικά από την τιμή του node τότε πηγαίνει στο δεξί παιδί του κόμβου και καλεί ξανά τη συνάρτηση **DeleteElement()** με το όρισμα του κόμβου αυτή τη φορά να είναι το δεξί παιδί.
    - Αλλιώς αν η λέξη είναι μικρότερη λεξικογραφικά από την τιμή του node τότε πηγαίνει στο αριστερό παιδί του κόμβου και καλεί ξανά τη συνάρτηση **DeleteElement()** με το όρισμα του κόμβου αυτή τη φορά να είναι το αριστερό παιδί.
    - Αν βρέθηκε η λέξη που θα διαγραφτεί, ελέγχεται αν υπάρχει ένα ή κανένα παιδί. Αν υπάρχει αποθηκεύεται το δεξί ή αριστερό παιδί (ανάλογα ποιο υπάρχει), διαγράφεται ο κόμβος της λέξης και στον πρόγονό του επιστρέφει τον δείκτη του παιδιού.
    - Αν βρέθηκε η λέξη που θα διαγραφτεί και ο κόμβος αυτός έχει δύο παιδιά, τότε καλείται η συνάρτηση **sInorder()** που θα επιστρέψει τον inorder successor του node. Επιπλέον αποθηκεύεται σε έναν pointer temporaryTree. Τα στοιχεία του θα

αποθηκευτούν στον κόμβο που θα διαγραφεί. Λόγω ίδιων κόμβων διαγράφουμε τον κόμβο με τιμή, ίδια αυτής της temporaryTree, του δεξιού παιδιού. Τέλος, η συνάρτηση επιστρέφει τον κόμβο.

- **SearchElement():** Δέχεται ως ορίσματα την λέξη που θα αναζητηθεί, καθώς και το node της ρίζας.
    - Αν βρει κόμβο που είναι κενός (NULL) τότε επιστρέφει τον κόμβο.
    - Αν ο κόμβος του ορίσματος περιέχει την προς αναζήτηση λέξη, τότε τον επιστρέφει.
    - Αν η λέξη είναι μεγαλύτερη λεξικογραφικά από την τιμή του node τότε πηγαίνει στο δεξί παιδί του κόμβου και καλεί ξανά τη συνάρτηση **SearchElement()** με το όρισμα του κόμβου αυτή τη φορά να είναι το δεξί παιδί.
    - Αλλιώς αν η λέξη είναι μικρότερη λεξικογραφικά από την τιμή του node τότε πηγαίνει στο αριστερό παιδί του κόμβου και καλεί ξανά τη συνάρτηση **SearchElement()** με το όρισμα του κόμβου αυτή τη φορά να είναι το αριστερό παιδί.
- **sInorder():** Δέχεται ως όρισμα έναν κόμβο.
    - Δημιουργεί καινούργιο αντικείμενο BinaryTree όπου αποθηκεύει την τιμή του κόμβου του ορίσματος.
    - Στη συνέχεια μέσω ενός while-loop πηγαίνει στο πιο αριστερά παιδί του δέντρου και στο τέλος το επιστρέφει.

AVL (AVLTree)

Η δομή του AVL αποτελείται από τις λειτουργίες εισαγωγής, διαγραφής και αναζήτησης, καθώς και από τις προσπελάσεις preorder, postorder και inorder. Αναλυτικότερα,

- **heightOfAVL():** δέχεται ως όρισμα ένα node και επιστρέφει το ύψος του.
- **calcDifHeight():** δέχεται ως όρισμα ένα node, υπολογίζει τη διαφορά ύψους μεταξύ του δεξιού και του αριστερού παιδιού του και την επιστρέφει.
- **maxValue():** δέχεται ως ορίσματα δύο ακέραιους αριθμούς και επιστρέφει τον μεγαλύτερο.
- **rightRotation():** δέχεται ως όρισμα ένα node στο οποίο θα εκτελεστεί η δεξιά περιστροφή.
    - Στον δείκτη root αποθηκεύεται η τιμή του αριστερού παιδιού του ορίσματος, ο οποίος δείκτης θα γίνει στη συνέχεια η ρίζα του υποδέντρου που θα περιστραφεί.
    - Στον δείκτη tempNode αποθηκεύεται η τιμή η τιμή του δεξιού παιδιού του ορίσματος, ο οποίος δείκτης θα γίνει στη συνέχεια η ρίζα.
    - Το root είναι η καινούργια ρίζα, η οποία δέχεται σαν δεξί παιδί το node του ορίσματος και σαν αριστερό παιδί το tempNode.
    - Βρίσκουμε τα ύψη των node που έγινε η περιστροφή σε συσχέτιση με το μέγιστο ύψος των παιδιών τους και αύξηση τους κατά ένα.
- **leftRotation():** δέχεται ως όρισμα ένα node στο οποίο θα εκτελεστεί η αριστερά περιστροφή.

- Στον δείκτη root αποθηκεύεται η τιμή του δεξιού παιδιού του ορίσματος, ο οποίος δείκτης θα γίνει στη συνέχεια η ρίζα του υποδέντρου που θα περιστραφεί.
- Στον δείκτη tempNode αποθηκεύεται η τιμή η τιμή του αριστερού παιδιού του ορίσματος, ο οποίος δείκτης θα γίνει στη συνέχεια η ρίζα.
- Το root είναι η καινούργια ρίζα, η οποία δέχεται σαν δεξί παιδί το tempNode και σαν αριστερό παιδί το node του ορίσματος.
- Βρίσκουμε τα ύψη των node που έγινε η περιστροφή σε συσχέτιση με το μέγιστο ύψος των παιδιών τους και αύξηση τους κατά ένα.

- **insertNode():** Δέχεται ως ορίσματα την λέξη που θα εισαχθεί καθώς και το node της ρίζας.
  - Αν η λέξη που έχει δοθεί ως όρισμα βρεθεί ήδη μέσα στην δομή τότε αυξάνει το πλήθος της συγκεκριμένης λέξης.
  - Αν το node που ελέγχεται είναι κενό (NULL) τότε δημιουργείται νέο αντικείμενο.
  - Αλλιώς καλείται η **insertNode()** με αναδρομή (η εισαγωγή γίνεται με βάσει τη λεξικογραφική σειρά. Η διαδικασία σταματάει αν βρεθεί η λέξη ή κενό node.
  - Αυξάνει το ύψος του node κατά ένα, καλώντας τις συναρτήσεις **maxValue()** και **heightOfAVL()** και επίσης καλεί την **calcDifHeight()** ώστε να υπολογίσει τη διαφορά του ύψους μεταξύ των δύο παιδιών.
  - Στη συνέχεια διακρίνουμε τις παρακάτω περιπτώσεις:
    - Αν η διαφορά είναι μεγαλύτερη του 1 και η λέξη του ορίσματος είναι μεγαλύτερη λεξικογραφικά από την τιμή του αριστερού κόμβου, δηλαδή είναι δεξιά του αριστερού παιδιού, τότε εκτελούμε μια αριστερά περιστροφή στο παιδί και στη συνέχεια μια δεξιά στον κόμβο.
    - Αν η διαφορά είναι μικρότερη του -1 και η λέξη του ορίσματος είναι μικρότερη λεξικογραφικά από την τιμή του δεξιού κόμβου, δηλαδή είναι αριστερά του δεξιού παιδιού, τότε εκτελούμε μια δεξιά περιστροφή στο παιδί και στη συνέχεια μια αριστερά στον κόμβο.
    - Αν η διαφορά είναι μεγαλύτερη του 1 και η λέξη του ορίσματος είναι μικρότερη λεξικογραφικά από την τιμή του αριστερού κόμβου, δηλαδή είναι αριστερά του αριστερού παιδιού, τότε εκτελούμε μια δεξιά περιστροφή.
    - Αν η διαφορά είναι μικρότερη του -1 και η λέξη του ορίσματος είναι μεγαλύτερη λεξικογραφικά από την τιμή του δεξιού κόμβου, δηλαδή είναι δεξία του δεξιού παιδιού, τότε εκτελούμε μια αριστερά περιστροφή.
- **deleteNode():** Δέχεται ως ορίσματα την λέξη που θα εισαχθεί καθώς και το node της ρίζας.
  - Αν το node που ελέγχεται είναι κενό (NULL) τότε το επιστρέφει.
  - Αν η λέξη είναι μεγαλύτερη λεξικογραφικά από την τιμή του node τότε πηγαίνει στο δεξί παιδί του κόμβου και καλεί ξανά τη συνάρτηση **deleteNode()** με το όρισμα του κόμβου αυτή τη φορά να είναι το δεξί παιδί.
  - Αλλιώς αν η λέξη είναι μικρότερη λεξικογραφικά από την τιμή του node τότε πηγαίνει στο αριστερό παιδί του κόμβου και καλεί ξανά τη συνάρτηση **deleteNode()** με το όρισμα του κόμβου αυτή τη φορά να είναι το αριστερό παιδί.
  - Αν βρέθηκε η λέξη, έχουμε τις παρακάτω περιπτώσεις:

        - Ο κόμβος που περιέχει τη λέξη έχει δύο παιδιά. Σε έναν δείκτη tempNode αποθηκεύεται ο δείκτης του πιο αριστερά του δεξιού παιδιού. Έπειτα στον κόμβο που θα διαγραφεί αντιγράφεται η τιμή του tempNode και διαγράφεται.
        - Ο κόμβος που περιέχει τη λέξη έχει ένα ή κανένα παιδί. Σε έναν δείκτη tempNode αποθηκεύεται η τιμή του παιδιού που υπάρχει, αλλιώς αν δεν υπάρχει αποθηκεύεται η τιμή NULL και αντιγράφεται το tempNode στον κόμβο που θα διαγραφεί.
    - Αφού διαγραφούν οι κόμβοι, αυξάνει το ύψος του node κατά ένα, καλώντας τις συναρτήσεις **maxValue()** και **heightOfAVL()** και επίσης καλεί την **calcDifHeight()** ώστε να υπολογίσει τη διαφορά του ύψους μεταξύ των δύο παιδιών.
    - Στη συνέχεια διακρίνουμε τις παρακάτω περιπτώσεις:
        - Αν η διαφορά είναι μεγαλύτερη του 1 και η διαφορά του ύψους με ρίζα το αριστερό παιδί είναι μικρότερη του 0, τότε εκτελούμε μια αριστερά περιστροφή στο παιδί και στη συνέχεια μια δεξιά στον κόμβο.
        - Αν η διαφορά είναι μικρότερη του -1 και η διαφορά του ύψους με ρίζα το δεξί παιδί είναι μεγαλύτερη του 0, τότε εκτελούμε μια δεξιά περιστροφή στο δεξί παιδί και στη συνέχεια μια αριστερά στον κόμβο.
        - Αν η διαφορά είναι μεγαλύτερη του 1 και η διαφορά του ύψους με ρίζα το αριστερό παιδί είναι μεγαλύτερη ή ίση του 0, τότε εκτελούμε μια δεξιά περιστροφή.
        - Αν η διαφορά είναι μικρότερη του -1 και η διαφορά του ύψους με ρίζα το δεξί παιδί είναι μικρότερη ή ίση του 0, τότε εκτελούμε μια αριστερά περιστροφή.
- **searchInAVL():** Η υλοποίηση της αναζήτησης είναι ίδια με αυτή του binaryTree.

Open Address HashTable (HashTable)

Η δομή του HashTable αποτελείται από τις λειτουργίες εισαγωγής και αναζήτησης, τις βοηθητικές συναρτήσεις **empty(), hashingFunction()** και την κλάση Slot. Αναλυτικότερα,

- **hashingFunction():** δέχεται ως όρισμα μια συμβολοσειρά. Ο αλγόριθμος κατακερματισμού που χρησιμοποιήθηκε είναι ο djb2, ο οποίος αναλύεται παρακάτω:
    - Αρχικοποίηση μεταβλητών, hash = 5381 και το μέγεθος της συμβολοσειράς.
    - Μέσα σε ένα for-loop για κάθε γράμμα της συμβολοσειράς δημιουργούνται μοναδικά κλειδιά.
    - Επιστρέφεται το hash%theSize έτσι ώστε να περιοριστούν οι τιμές στο διάστημα [0, theSize)
- **searchHashTable():** δέχεται ως όρισμα τη λέξη που θα αναζητηθεί.
    - Αρχικοποιούνται οι μεταβλητές.
    - Σε μια do-while-loop όσο δεν έχει βρεθεί η λέξη και δεν έχει προσπελαστεί όλος ο πίνακας, γίνεται έλεγχος για το αν το κλειδί που δόθηκε από τη συνάρτηση **hashingFunction()** δείχνει σε NULL κελί.
        - Αν δείχνει σε NULL, τότε δημιουργούμε ένα αντικείμενο Slot, αποθηκεύουμε τη λέξη και αλλάζει η τιμή της λογικής μεταβλητής σε true έτσι ώστε να ολοκληρωθεί το do-while-loop.

        - Αν ναι ελέγχουμε αν μέσα στο κελί βρίσκεται η λέξη προς αναζήτηση. Αν υπάρχει τότε αλλάζει η τιμή της λογικής μεταβλητής σε true έτσι ώστε να ολοκληρωθεί το do-while-loop, σε αντίθετη περίπτωση προχωράμε στο επόμενο κελί.
- **insertHashTable():** δέχεται ως όρισμα τη λέξη που θα εισαχθεί.
    - Αρχικοποιούνται οι μεταβλητές.
    - Σε μια do-while-loop όσο δεν έχει βρεθεί η λέξη και δεν έχει προσπελαστεί όλος ο πίνακας, γίνεται έλεγχος για το αν το κλειδί που δόθηκε από τη συνάρτηση **hashingFunction()** δείχνει σε NULL κελί.
        - Αν δείχνει σε NULL, τότε δημιουργούμε ένα αντικείμενο Slot, αποθηκεύουμε τη λέξη και αλλάζει η τιμή της λογικής μεταβλητής σε true έτσι ώστε να ολοκληρωθεί το do-while-loop.
        - Αν ναι ελέγχουμε αν μέσα στο κελί βρίσκεται η λέξη προς αναζήτηση. Αν υπάρχει τότε κάνει break και βγαίνει από το do-while-loop, σε αντίθετη περίπτωση προχωράμε στο επόμενο κελί, αλλά με κλειδί ίσο με (i+1)%theSize έτσι ώστε όταν είναι στο τελευταίο κελί του Hash Table να επισκεφθεί στη συνέχεια το πρώτο κελί.
- **empty():** δέχεται ως όρισμα ένα κλειδί και ελέγχει αν το κελί του κλειδιού είναι κενό.

Σημείωση:
Για κάθε κλάση .cpp δημιουργήθηκαν τα απαραίτητα .h αρχεία που περιέχουν τις δηλώσεις για κάθε συνάρτηση που χρησιμοποιείται. Ακόμη το αρχείο που διαβάζει ονομάζεται **file.txt.**