



ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάθημα **(NCO-02-03) Δομές Δεδομένων**

Διδάσκοντες **Παπαδόπουλος Απόστολος, Βράκας Δημήτριος**

Εργασία **Εργασία 2021** (κατ. ημερ/νία 20/06/2021)

Φοιτητές	3865 Μπαλακτσής Χρήστος
	3990 Παναγιωτόπουλος Νέρων - Μιχαήλ

Τεχνική Έκθεση (documentation essay)

1 Εισαγωγή

Η παρούσα εργασία υλοποιήθηκε στα πλαίσια του μαθήματος του 2ου εξαμήνου «**Δομές Δεδομένων**» του ΠΠΣ του τμήματος Πληροφορικής του ΑΠΘ.

Η εργασία αποτελείται από δύο τμήματα:

- αρχεία **πηγαίου κώδικα** (.cpp, .h) σε γλώσσα **C++** (v.14), για την υλοποίηση συγκεκριμένων δομών δεδομένων και λειτουργιών επ' αυτών, όπως περιγράφονται στην εκφώνηση του προβλήματος, και
- την παρούσα **Έκθεση αναφοράς**, για την περιγραφή των δομών και των τρόπων λειτουργίας βάσει των οποίων αναπτύχθηκαν.

2 Ανάλυση Προβλήματος

2.1 Στόχος

Ζητούμενο του προβλήματος είναι η υλοποίηση πέντε (5) δομών δεδομένων, όπως αναφέρονται εκτενώς παρακάτω, στις οποίες αποθηκεύονται ως δεδομένα τύπου string όλες οι λέξεις -χωρίς διπλότυπες αντιγραφές- ενός δοθέντος κειμένου σε αρχείο επέκτασης .txt.

Οι δομές που αναπτύχθηκαν είναι οι εξής πέντε:

- αταξινόμητος πίνακας (**Unordered Array**)
- ταξινομημένος πίνακας (**Ordered Array**)
- απλό δυαδικό δένδρο αναζήτησης (**Binary Search Tree**)
- δένδρο αναζήτησης AVL (**AVL Binary Search Tree**)
- πίνακας κατακερματισμού ανοικτής διεύθυνσης (**Hash Table**)

Όλες οι δομές αποθηκεύουν την ίδια πληροφορία, αλλά τη χειρίζονται με διαφορετικό τρόπο και κάθε τύπος δομής, επιδέχεται διαφορετικές συναρτήσεις και τεχνικές για την υλοποίηση των βασικών λειτουργιών χειρισμού των δομών δεδομένων.

2.2 Συμβάσεις

Για την εισαγωγή των διαφόρων, μη διπλότυπων λέξεων του κειμένου προϋποτίθεται μια στοιχειώδης διαδικασία προ-επεξεργασίας των λέξεων αυτών, προσανατολισμένη στην αποφυγή αποθήκευσης **νοηματικά διπλότυπων όρων**. Αναλυτικότερα, ορίζουμε:

- **ομοιότητα** μεταξύ πεζών και κεφαλαίων γραμμάτων, δηλαδή δεν θεωρούμε διαφορετικές λέξεις όσες συνίστανται από τα ίδια λατινικά γράμματα και την ίδια διάταξη, ακόμη και αν διαφέρουν ως προς την μορφολογική τους αναπαράσταση (κεφαλαία ή πεζά). Παραδείγματος χάριν, ο όρος "this" θεωρούμε ότι είναι ισοδύναμος με τους όρους: This, this, tHiS, thiS κ.ο.κ.

- **μη διάκριση** των περιφραστικών-σύνθετων λέξεων χωρισμένων με παύλα (-) σε τόσες λέξεις όσοι και οι όροι που περιλαμβάνει η σύνθετη αυτή λέξη. Παραδείγματος χάριν, ο όρος "mother-in-law" θεωρούμε ότι αποτελεί από μία λέξη (συμβατικά: "motherinlaw").
- **αδιάφορους όρους** κάθε σύμβολο που δεν ανήκει στο λατινικό αλφάβητο, όπως αριθμοί (π.χ. -5, 1,700) και ειδικά σύμβολα (π.χ. ., !, @, "). Οι αδιάφοροι όροι δεν προσμετρώνται ως λέξεις του κειμένου, ενώ, σε τυχούσες παράδοξες τοποθετήσεις αυτών εντός λέξεων (π.χ. th1s), οι λέξεις θεωρούνται έγκυρες και προσμετρώνται μόνο μετά από αγνόση/αφαίρεση των όρων αυτών.

3 Περιγραφή Δομών Δεδομένων

Κάθε δομή αποτελείται από τουλάχιστον δύο αρχεία: ένα **αρχείο βιβλιοθήκης**, όπου περιλαμβάνεται η **κλάση** που αναπαριστά την αντίστοιχη δομή μαζί με τις υπογραφές των μεθόδων της, και ένα **αρχείο υλοποίησης**, όπου υλοποιούνται οι **μέθοδοι** αυτές. Γενικά, κάθε δομή/κλάση περιέχει τις εξής στοιχειώδεις λειτουργίες επί του αποθηκευμένου περιεχομένου τους: **εισαγωγή**, **διαγραφή*** και **αναζήτηση** συμβολοσειράς. Ο χειρισμός των λέξεων ως στοιχεία της κάθε δομής υλοποιήθηκε με τη βιβλιοθήκη **string** και κάθε λέξη του κειμένου αποθηκεύεται ως αφηρημένος τύπος δεδομένων **string**. Η χρήση των αναφορικών ορισμάτων, όπου ήταν θεμιτό και δυνατό, εξυπρετεί στην ταχύτερη λειτουργία του προγράμματος.

3.1 Αταξινόμητος πίνακας

Η δομή του αταξινόμητου πίνακα αφορά στα αρχεία UnorderedArray.h και UnorderedArray.cpp. Το αρχείο βιβλιοθήκης (.h) περιλαμβάνει τη (δημόσια) κλάση UnorderedArray.

```
class UnorderedArray {
protected:
    int step;
    string *data;
    int *num;
    long size;
    long current_size;
    bool Search_help(const string &, long &);

public:
    UnorderedArray();
    ~UnorderedArray();
    void insert(const string &);
    void insertUnique(const string &, int);
    int search(const string &);
    bool deleteWord(const string &);
    int getSize() const;
    int getNum(long) const;
    string getData(long) const;
    void setNum(long, int);
};
```

Κώδικας 1. Η κλάση UnorderedArray.

Μέγεθος πίνακα & Γενικές Λειτουργίες

Ο αταξινόμητος πίνακας αποτελείται από δύο παράλληλους δυναμικούς πίνακες: **string *data**, για την αποθήκευση των λέξεων και **int *num**, για την αποθήκευση του πλήθους εμφανίσεων κάθε λέξης.

Το μέγεθος του πίνακα (**long size**) **προσαυξάνεται κατά 5000** στοιχεία, μόλις συμπληρώνονται οι προηγούμενες $5000 \cdot k$, $k = 1, \dots, \text{θέσεις του}$. Για να πραγματοποιηθεί αυτό, επειδή χρησιμοποιούνται συνεχόμενες θέσεις

* Ορίζουμε ως διαδικασία **διαγραφής** μιας συμβολοσειράς από μια δομή τη μέθοδο, κατά την οποία απαλείφεται από τη δομή τόσο η συμβολοσειρά όσο και ο αριθμός εμφανίσεων αυτής στο αρχείο εισόδου. Η μέθοδος επηρεάζει μόνο τη δομή πάνω στην οποία εφαρμόζεται και δε μεταβάλλει το περιεχόμενο των άλλων δομών ή του αρχείου κειμένου. Η μέθοδος της διαγραφής δεν υλοποιήθηκε για τη δομή του πίνακα κατακερματισμού, όπως ζητήθηκε.

στη μνήμη, πρέπει σε κάθε επέκταση του πίνακα να **αντιγράφονται** τα στοιχεία του σε έναν νέο αταξινόμητο πίνακα (ενν. έναν για τις λέξεις και έναν για τις εμφανίσεις των λέξεων). Η αντιγραφή, ωστόσο, ολόκληρης της δομής είναι μια χρονοβόρα διαδικασία, ενώ η συχνή εφαρμογή της μειώνει σημαντικά την αποδοτικότητα του προγράμματος. Γι' αυτόν τον λόγο, επιλέχθηκε να πραγματοποιείται μετά τη συμπλήρωση 5000 στοιχείων (ανά πίνακα), έναντι της επέκτασης του πίνακα κατά ένα στοιχείο σε κάθε εισαγωγή. Ο αριθμός «5000» επιλέχθηκε με βάση τις συνολικές μοναδικές λέξεις του κειμένου, αλλά είναι ενδεικτικός.

Για την προσαρμοσμένη/προσδιορισμένη δημιουργία αντικειμένων της κλάσης `UnorderedArray` συντάχθηκε ο **κατασκευαστής** `UnorderedArray()`, στον οποίο αρχικοποιούνται οι δείκτες των δυναμικών πινάκων λέξεων και εμφανίσεων, καθώς και οι μεταβλητές που αφορούν το **μέγεθος** του χροσιμοποιημένου και του δεσμευμένου πίνακα, `size` και `current_size`, αντίστοιχα, σε μηδέν, ενώ το **βήμα επέκτασης** ορίζεται στην αμετάβλητη τιμή `step=5000` θέσεις. Ακόμη, αναπτύχθηκε ο **αποκατασκευαστής** `~UnorderedArray()`, που αναλαμβάνει να αποδεσμεύει τη μνήμη που καταλαμβάνουν οι πίνακες, όταν είναι απαραίτητο.

Για τον χειρισμό των διαφόρων ιδιωτικών-προστατευμένων μελών της κλάσης, συντάχθηκαν οι απαραίτητοι `setters` και `getters` των τιμών των μεταβλητών αυτών:

```
› int getSize() const;           // πρόσβαση στο τρέχον συνολικό μέγεθος του πίνακα
› int getNum(long) const;        // πρόσβαση στο num[<όρισμα>]
› string getData(long) const;    // πρόσβαση στο data[<όρισμα>]
› void setNum(long, int);        // ορισμός της τιμής num[<όρισμα>]
```

Εισαγωγή στοιχείων

Η μαζική εισαγωγή των λέξεων του αρχείου κειμένου πραγματοποιείται με τη σειριακή κλήση της μεθόδου `insert(const string &)`, η οποία δέχεται μια συμβολοσειρά (λέξη), την αναζητεί **σειριακά** (χροσιμοποιώντας την ιδιωτική μέθοδο (σ.σ. συνάρτηση) `Search_help(const string &, long &)`) εντός της δομής του αταξινόμητου πίνακα `data[]` και:

- › αν **υπάρχει** ήδη στη θέση `pos` του `data[]`, τότε προσαυξάνεται κατά ένα το πλήθος εμφανίσεων (`num[pos] += 1`).
- › αν **δεν υπάρχει**, τότε καλείται η μέθοδος `insert_Unique`, η οποία αναλαμβάνει να εισάγει τη νέα λέξη στον πίνακα `data[]` και να ορίσει ως πλήθος εμφανίσεων αυτής στον πίνακα `num[]` το 1 (ενν. ότι η εισαγωγή πραγματοποιείται στην πρώτη κενή θέση του πίνακα, και αν αυτός είναι συμπληρωμένος, πρώτα επεκτείνεται το μέγεθός του κι έπειτα εισάγεται η λέξη, όπως περιεγράφηκε παραπάνω).

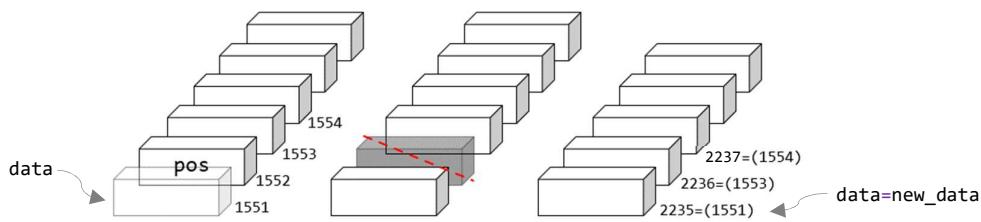
Η μέθοδος `insertUnique(const string &, int)` υλοποιήθηκε προκειμένου να εξυπηρετήσει τόσο τις ανάγκες της μεθόδου `insert` για την εισαγωγή νέων λέξεων στη δομή (**βοηθητική χρήση**) όσο και για την ευκολότερη από πλευράς χρόνου **εκσφαλμάτωση** (debugging) του κώδικα κατά τον σχεδιασμό του προγράμματος. Η εν λόγω μέθοδος προϋποθέτει έλεγχο με άλλο-εξωτερικό τρόπο των διπλότυπων όρων, δέχεται ως όρισμα την πρωτοεμφανιζόμενη νέα λέξη και έναν ακέραιο αριθμό εμφανίσεων και αναλαμβάνει να τα εισάγει στη δομή.

Διαγραφή στοιχείου

Η διαγραφή μιας λέξης από τη δομή του αταξινόμητου πίνακα συνεπάγεται τη διαγραφή/παράλειψη των κόμβων/στοιχείων του αταξινόμητου πίνακα λέξεων και του παράλληλου πίνακα αντίστοιχων εμφανίσεων, που αφορούν τη λέξη αυτή, διαγράφονται όλες οι εμφανίσεις της. Η διαδικασία αυτή πραγματοποιείται μέσω της μεθόδου `deleteWord(const string &)` που εφαρμόζεται επί της κλάσης `UnorderedArray`. Η μέθοδος δέχεται ως όρισμα μια συμβολοσειρά (`string`) και αναλαμβάνει να αναζητήσει σειριακά -χροσιμοποιώντας τη μέθοδο (σ.σ. συνάρτηση) `Search_help(const string &, long &)` για τη γνώση της ύπαρξης και της θέσης για- τη συμβολοσειρά αυτή μέσα στον πίνακα `data[]` και:

- › αν **δεν υπάρχει**, τότε δεν πραγματοποιείται καμία αλλαγή στο περιεχόμενο της δομής και η μέθοδος επιστρέφει **False** (Boolean value).
- › αν **υπάρχει**, έστω στη θέση `pos`, τότε αντιγράφει τα περιεχόμενα των δύο πινάκων της δομής (`data[]` και `num[]`) σε δύο νέους πίνακες (`new_data[]` και `new_num[]`), μεγέθους **κατά 1 στοιχείο λιγότερο** των αρχικών, παραβλέποντας το στοιχείο (λέξη) για το οποίο κλήθηκε η μέθοδος, δηλαδή, κατά την αντιγραφή,

παραλείπονται τα `data[pos]` και `num[pos]`. Μετά το πέρας της σύνθεσης των νέων πινάκων, διαγράφονται από τη μνήμη οι παλιοί (`delete[]`) και οι δείκτες `*data` και `*num` αντιστοιχίζονται στις διευθύνσεις των νέων. Με την ολοκλήρωση της συνολικής διαδικασίας, επιστρέφεται **True** (Boolean value).



Σχήμα 1. Διαγραφή του στοιχείου της θέσης `pos` σε δυναμικό πίνακα με συμβατική αναπαράσταση. Η τρίτη συστοιχία αντιστοιχεί σε άλλο σημείο της μνήμης, όπου αντιγράφηκαν τα επιθυμητά κελιά που αντιστοιχούσαν στην προηγούμενη μορφή της δομής.

Αναζήτηση στοιχείου

Η αναζήτηση μιας συγκεκριμένης λέξης που **εν δυνάμει** υπάρχει στο κείμενο και κατ' επέκταση στη δομή του αταξινόμητου πίνακα, πραγματοποιείται με την κλήση της μεθόδου `search(const string &)`, η οποία δέχεται μια συμβολοσειρά και την αναζητά **σειριακά** στον πίνακα `data[]` και **επιστρέφει το πλήθος εμφανίσεων** της λέξης αυτής στο αρχείο κειμένου. Αν δεν υπάρχει καμία φορά ή ο πίνακας είναι κενός, η μέθοδος επιστρέφει την τιμή **μηδέν** (0). Υπενθυμίζεται ότι η σειριακή αναζήτηση είναι η μοναδική τεχνική αναζήτησης που μπορεί να χρησιμοποιηθεί στους αταξινόμητους πίνακες και αποτελεί διαδικασία υψηλού κόστους ($O(n)$), για αυτό και η λειτουργία της μεθόδου είναι χρονοβόρα.

Η σειριακή αναζήτηση συνοψίζεται στη γραμμική, διαδοχική προσπέλαση και σύγκριση καθενός στοιχείου του πίνακα με το αναζητούμενο **κλειδί αναζήτησης** (το όρισμα), μέχρι να εντοπιστεί η αναζητούμενη συμβολοσειρά/λέξη. Στη χειρότερη περίπτωση, που η λέξη δεν υφίσταται, θα πρέπει να προσπελαστεί ολόκληρος ο πίνακας. Παράλληλα, για τις ανάγκες **μεμονωμένης εισαγωγής** και **διαγραφής** λέξης, κρίθηκε αναγκαία η ανάπτυξη μιας **βοηθητικής μεθόδου αναζήτησης**, της `Search_help(const string &, long &)`, η οποία δέχεται μια συμβολοσειρά/λέξη, την αναζητά σειριακά (εφόσον ο πίνακας `data[]` δεν είναι κενός) και **επιστρέφει αναφορικά** τη **θέση** μέσα στον πίνακα, που είναι αποθηκευμένη η συμβολοσειρά, καθώς επίσης, και την τιμή **True** ή **False**, σχετικά με το αν βρέθηκε ή όχι εντός του πίνακα, αντίστοιχα. Η μέθοδος αυτή αποτελεί προστατευμένο μέλος της κλάσης, διότι δεν προορίζεται για απευθείας χρήση σε κοινές συναρτήσεις ή στην `main`, αλλά μόνο για εκ περιτροπής ανάγκη αναζήτησης λέξης, με απαραίτητη τη **γνώση της θέσης** της, εφόσον αυτή υπάρχει.

3.2 Ταξινομημένος πίνακας

Η δομή του ταξινομημένου πίνακα αποτελείται από τα αρχεία `orderedArray.h` και `orderedArray.cpp`. Το Header αρχείο περιλαμβάνει τη κλάση `orderedArray`. Το header αρχείο περιλαμβάνει τη κλάση `orderedArray` και την ιδιωτική κλάση `Cell`, η οποία περιλαμβάνει τα δεδομένα ενός στοιχείου της δομής (λέξη και εμφανίσεις).

```
class orderedArray {
private:
    class Cell {
        public:
            string word;
            int occurrences = 0;
    };
    Cell *table;
    long size;
    bool binSearch(const string &word, long &pos);
    void quicksort(long start, long end);
    void swap(Cell &a, Cell &b);
public:
```

```

        orderedArray();
        void copyFromUnordered(string *newData, int *newNum, long ar-
raySize);
        void insert(const string& word);
        void insertUnique(const string& word, int occurrences);
        int search(const string& word);
        void remove(const string& word);
        long getSize() const;
    };

```

Κώδικας 2. Η κλάση orderedArray.

Εισαγωγή στοιχείων

Η κλάση παρέχει τρεις συναρτήσεις για την εισαγωγή στοιχείων.

- **insert(const string& word):** Η πρώτη και πιο απλή (προς τον χρήστη) συνάρτηση προορίζεται για εισαγωγές μικρού-μεσαίου αριθμού λέξεων. Δέχεται σαν όρισμα μια μεταβλητή τύπου string και εκτελεί δυαδική αναζήτηση στα περιεχόμενα του πίνακα, ώστε να ελέγχει εάν είχε καταχωριθεί ήδη στο παρελθόν. Αν βρεθεί, θα αυξήσει τον μετρητή εμφανίσεων της κατά 1. Σε περίπτωση μη εύρεσής της, θα εκτελέσει μια λειτουργία όμοια της insertion-sort, όπου θα αναζητηθεί η θέση και στην οποία θα άνηκε η λέξη ώστε να είναι ταξινομημένος ο πίνακας data και θα γίνει ολίσθηση των στοιχείων σε υψηλότερες θέσεις κατά μια μονάδα, ενώ στην και τα καταχωριθούν η λέξη και το “1” στους πίνακες των λέξεων και εμφανίσεων αντίστοιχα.
- **insertUnique(const string& word, int occurrences):** Η συνάρτηση αυτή υποθέτει πως ο χρήστης έχει παράξει αλγόριθμο για την αντιστοίχιση των λέξεων με τις εμφανίσεις, συνεπώς η insertUnique δέχεται ως όρισμα έναν ακέραιο “occurrences” που αντιπροσωπεύει τις εμφανίσεις και τα καταχωρεί στους πίνακες με την ίδια μέθοδο όπως στην insert, παρακάμπτοντας την προσαύξηση στον πίνακα num σε περίπτωση μη μοναδικότητας της λέξεως.
- **copyFromUnordered(string *newData, int *newNum, long arraySize):** Σκοπός αυτής της συνάρτησης είναι η παραγωγή ταξινομημένου πίνακα από αταξινόμητο, με τον ταχύτερο δυνατό τρόπο, συνεπώς προορίζεται για σπάνιες χρήσεις, όπου ο όγκος των δεδομένων προκαλεί σημαντικές καθυστερήσεις στον χρόνο εκτέλεσης του προγράμματος. Αυτή η συνάρτηση δέχεται ως ορίσματα pointer στα στοιχεία ενός Unordered Array, όπως και το μέγεθός του. Ύστερα αρχικοποιεί τη δομή “table” τύπου Cell του αντικειμένου, αντιγράφοντας τα στοιχεία από τα ορίσματα και εκτελεί τον αλγόριθμο ταξινόμισης quicksort στις λέξεις, με τη χρήση της ιδιωτικής συνάρτησης quicksort().

Διαγραφή στοιχείων

Η διαγραφή στοιχείων από τη δομή γίνεται μέσω της συνάρτησης remove(const string& word). Δέχεται ως όρισμα τη λέξη η οποία πρέπει να αφαιρεθεί από την δομή και εκτελεί δυαδική αναζήτηση ώστε να ελέγχει εάν υπάρχει εντός του πίνακα data. Εάν δεν υπάρχει, η συνάρτηση τερματίζει τη λειτουργία της με την εντολή return(). Εφόσον βρεθεί, θα επιστραφεί η θέση k και θα γίνει ολίσθηση όλων των στοιχείων σε θέση > k κατά μια μονάδα προς τα κάτω και θα μικρύνουν τα μεγέθη των πινάκων κατά ένα αντίστοιχα.

Αναζήτηση στοιχείων

Η αναζήτηση γίνεται μέσω της συνάρτησης search(const string& word) με όρισμα τη λέξη προς αναζήτηση και επιστρέφει ως ακέραιο τις εμφανίσεις του κλειδιού. Η συνάρτηση καλεί την ιδιωτική υλοποίηση του αλγόριθμου της δυαδικής αναζήτησης, με μέσο όρο απόδοσης $O(\log(n))$, η οποία επιστρέφει Boolean ανάλογα με το αν βρέθηκε το κλειδί και (μέσω κλήσης με αναφορά) την θέση του στοιχείου - pos. Ύστερα η public search επιστρέφει 0 αν δεν βρέθηκε η λέξη ή num[pos] που αντιπροσωπεύει τις καταγεγραμμένες εμφανίσεις της.

3.3 Δυαδικό Δένδρο Αναζήτησης

Η δομή του απλού δυαδικού δένδρου αναζήτησης (ΔΔΑ) αφορά στα αρχεία BSTree.h και BSTree.cpp. Το αρχείο βιβλιοθήκης (.h) περιλαμβάνει τη (δημόσια) κλάση BSTree, καθώς και το struct BTNode, το οποίο παριστάνει κόμβους ενός δυαδικού δένδρου.

```
class BSTree {  
protected:  
    BTNode *root;  
    void insert(BTNode*, const string &);  
    long getHeight(BTNode*);  
    void deleteBST(BTNode*);  
    void inOrder(BTNode*);  
    void preOrder(BTNode*);  
    void postOrder(BTNode*);  
    void deleteBST();  
public:  
    BSTree();  
    ~BSTree();  
    void insert(const string &);  
    bool deleteWord(const string &);  
    int search(const string &);  
    long getHeight();  
    void inOrder();  
    void preOrder();  
    void postOrder();  
};
```

Κώδικας 3. Η κλάση BSTree.

```
struct BTNode {  
    string data;  
    int num;  
    BTNode *left;  
    BTNode *right;  
    BTNode() {  
        left = nullptr;  
        right = nullptr;  
    }  
    BTNode(const string &word) {  
        data = word;  
        num = 1;  
        left = nullptr;  
        right = nullptr;  
    }  
    BTNode(const string &word, BTNode *l, BTNode  
        num = 1;  
        data = word;  
        right = r;  
        left = l;  
    };
```

Κώδικας 4. Η δομή κόμβου ΔΔΑ BTNode.

Περιγραφή της δομής BTNode

Για τη σύνθεση του (απλού) δυαδικού δένδρου αναζήτησης, κρίθηκε αναγκαία η δημιουργία μιας δομής (struct) για την **αναπαράσταση** κάθε **κόμβου**-στοιχείου του.

Ένας κόμβος περιέχει τη λέξη που αποθηκεύεται σε αυτόν (string data), το πλήθος εμφανίσεων αυτής εντός του αρχείου κειμένου της εισόδου (long num), καθώς και δύο δείκτες σε κόμβους (BTNode *left, *right). Η ύπαρξη του δεξιού και του αριστερού δείκτη εξυπηρετεί τη διασύνδεση των κόμβων στη δενδρική δομή, υπακούοντας στην **αρχή σύνθεσης του δένδρου αναζήτησης**, κατά την οποία κάθε κόμβος του δένδρου επιτρέπεται να έχει το πολύ δύο υποκόμβους-παιδιά (child nodes), ένα αριστερό κι ένα δεξιό. Το struct BTNode περιέχει, ακόμη, τρεις **κατασκευαστές** που αναλαμβάνουν να κατασκευάσουν έναν νέο κόμβο, τοποθετώντας (ή και όχι, αν πρόκειται για τον κενό κατασκευαστή) την επιθυμητή λέξη και αρχικοποιώντας το πλήθος εμφανίσεών της σε μονάδα.

Γενικές Λειτουργίες

Για την προσαρμοσμένη/προσδιορισμένη δημιουργία αντικειμένων της κλάσης BSTree συντάχθηκε ο **κατασκευαστής** BSTree(), με τον οποίο η ρίζα (*root) θέτεται να δείχνει σε NULL. Ακόμη, αναπτύχθηκε ο **αποκατασκευαστής** ~BSTree(), που αναλαμβάνει να αποδεσμεύει τη μνήμη που καταλαμβάνουν οι κόμβοι του δένδρου, όταν είναι απαραίτητο, με τη χρήση της συνάρτησης deleteBST(), η οποία διαγράφει κάθε κόμβο μέσω της προστατευμένης συνάρτησης deleteBST(BTNode*). Τέλος, η μέθοδος getHeight() -και η αντίστοιχη προστατευμένη επιστρέφει το **ύψος του δένδρου**, δηλαδή το πλήθος των κόμβων της μεγαλύτερης διαδρομής από τη ρίζα μέχρι κάποιο φύλλο. Λειτουργεί αναδρομικά, με δεικτοδοτούμενη προσπέλαση των κόμβων του αριστερού και δεξιού υποδένδρου κάθε κόμβου, προκειμένου να εντοπιστεί το μεγαλύτερο μονοπάτι.

Εισαγωγή στοιχείων

Η εισαγωγή κάθε λέξης στη δενδρική δομή πραγματοποιείται μέσω της μεθόδου insert(const string &), η οποία δέχεται μια συμβολοσειρά/λέξη του κειμένου και αναλαμβάνει να προσπελάσει τους κόμβους της διαδρομής εντός του δένδρου που η λέξη θα έπρεπε να ακολουθήσει, καταλήγοντας στον κατάλληλο κόμβο που πρέπει να

αποθηκευτεί. Έτσι, αναζητεί ταυτόχρονα τόσο τη θέση που πρέπει να τοποθετηθεί η λέξη, όσο και την ύπαρξη της λέξης εντός της δομής. Επομένως, προσεγγίζοντας την κατάλληλη θέση για τη λέξη:

- › αν **υπάρχει ήδη** σε αυτό το σημείο η λέξη, τότε απλά ενημερώνεται το πλήθος εμφανίσεων αυτής, προσαυξανόμενο κατά 1 (`tNode->num+=1`).
- › αν **δεν υπάρχει**, τότε τοποθετείται σε νέο κόμβο, ο οποίος θα αποτελεί είτε αριστερό ή δεξί παιδί του προηγούμενου κόμβου.

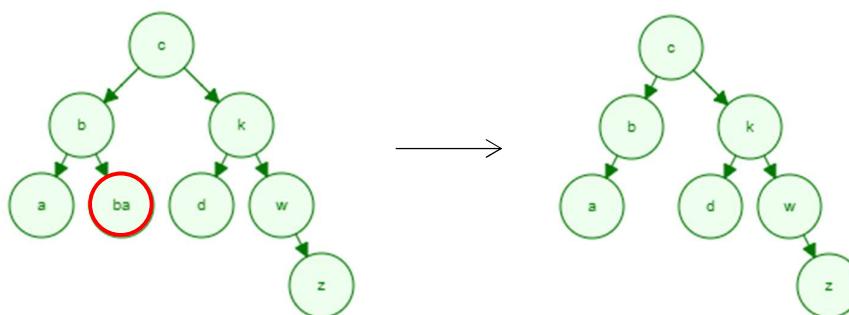
Το **κριτήριο** καθορισμού της **κατεύθυνσης** (δεξί-αριστερό) του κόμβου-παιδί ορίζεται από την **αρχή της σύνθεσης του δυαδικού δένδρου αναζήτησης**, κατά την οποία οι κόμβοι του δένδρου είναι ταξινομημένοι κατά αύξουσα σειρά από τα αριστερά προς τα δεξιά. Δηλαδή, για κάθε κόμβο *A* που βρίσκεται αριστερά ενός άλλου κόμβου *B* και είναι παιδί ενός άλλου κόμβου *K*, ισχύει: *A < K < B*. Η σχέση διάταξης «<» αναφέρεται στη **σύγκριση των λέξεων** που περιέχονται σε καθένα από τους κόμβους αυτούς. Η περίπτωση «≤» δεν λαμβάνεται υπόψιν, αφού για την ισότητα, δεν προκύπτει αποθήκευση νέας λέξης (δηλαδή, δημιουργία νέου κόμβου-παιδιού), αλλά ενημερώνεται το πλήθος (`num`) στον κόμβο της λέξης αυτής.

Πρακτικά, η διαδικασία που περιγράφεται παραπάνω υλοποιείται με τη βοήθεια της προστατευμένης μεθόδου `insert(BTNode*, const string &)`. Συγκεκριμένα, η κλήση της `insert(const string &)` ενεργοποιεί την προηγούμενη ομώνυμη συνάρτηση με ορίσματα τη ρίζα `root` και τη συμβολοσειρά που δόθηκε από την `main`. Η `insert(BTNode*, const string &)`, λοιπόν, αναλαμβάνει να διατρέξει τη δενδρική δομή κόμβο προς κόμβο (χρησιμοποιώντας **αναδρομική κλήση** με ορίσματα τους κόμβους-παιδιά των τρεχόντων κόμβων), ακολουθώντας τη διαδρομή που υποδεικνύεται από τις ανισοτικές σχέσεις των λέξεων.

Διαγραφή στοιχείων

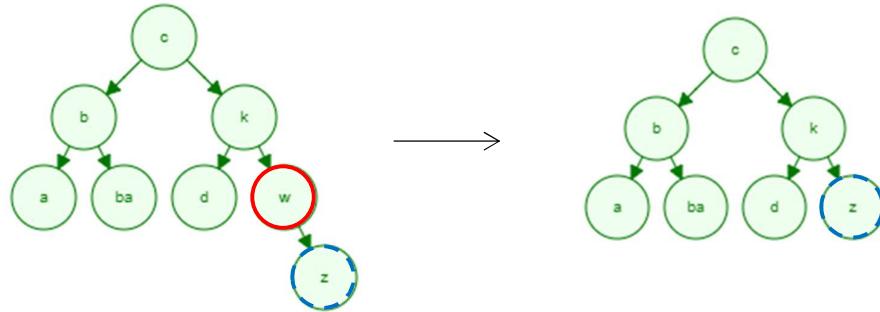
Η διαγραφή μιας λέξης από τη δομή του δυαδικού δένδρου αναζήτησης συνεπάγεται τη διαγραφή/παράλειψη του κόμβου που περιέχει τη αυτή τη συμβολοσειρά και το πλήθος των εμφανίσεων της εντός του κειμένου. Η διαδικασία αυτή πραγματοποιείται μέσω της μεθόδου `deleteWord(const string &)` που εφαρμόζεται επί της κλάσης `BTTree`. Η μέθοδος δέχεται ως όρισμα μια συμβολοσειρά και ξεκινάει τη δεικτοδοτούμενη αναζήτησή της εντός της δενδρικής δομής και:

- › αν **δεν υπάρχει**, επιστρέφει ***False*** (Boolean Value).
- › αν **υπάρχει** σε έναν κόμβο, έστω *p*, τότε η επιστρέφει ***True*** (Boolean Value), αφότου ελέγχει αν και πόσους κόμβους-παιδιά έχει και:
 - » αν **δεν έχει παιδιά**, τότε απλά διαγράφει τον κόμβο-**φύλλο** (`delete p`) και θέτει σε `nullptr` το δείκτη του κόμβου-γονέα του *p* που έδειχνε στον *p*.



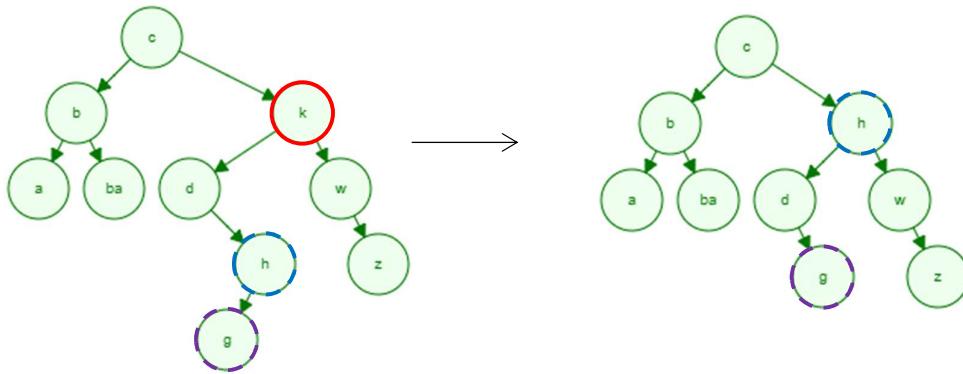
Σχήμα 2. Διαγραφή του κόμβου-φύλλου.

- » αν έχει **ένα παιδί**, έστω τον κόμβο *c*, τότε ο *c* αντικαθιστά τον *p*, δηλαδή ο δείκτης του γονέας του *p* που έδειχνε στον *p*, τίθεται να δείχνει στον *c*.



Σχήμα 3. Διαγραφή του κόμβου με ένα παιδί.

- » αν έχει **δύο παιδιά**, τότε ο κόμβος p **αντικαθίσταται** πάντα από το **μεγαλύτερο στοιχείο του αριστερού** υποδένδρου, προκειμένου να τηρηθεί η αρχή της σύνθεσης του δένδρου, και ο p πλέον έχει το πολύ ένα (αριστερό) παιδί, το οποίο μεταβαίνει ένα επίπεδο υψηλότερα (παίρνει τη θέση που κατείχε το στοιχείο-αντικαταστάτης του p).



Σχήμα 4. Διαγραφή του κόμβου με δύο παιδιά.

Αναζήτηση στοιχείου

Η αναζήτηση μιας συγκεκριμένης λέξης που **εν δυνάμει** υπάρχει στο κείμενο και κατ' επέκταση στη δομή του δυαδικού δένδρου αναζήτησης πραγματοποιείται με την κλήση της μεθόδου `search(const string &)`, η οποία δέχεται μια συμβολοσειρά, εκτελεί δεικτοδοτούμενη **δυαδική αναζήτηση** στους κόμβους του δένδρου με βάση το **κριτήριο κατεύθυνσης** που περιγράφηκε στην υποενότητα εισαγωγής στοιχείου στη δομή και **επιστέφει το πλήθος εμφανίσεων** της λέξης αυτής στο αρχείο κειμένου, όπως έχει αυτό αποθηκευτεί στον κόμβο που περιέχει τη συμβολοσειρά αυτή. Αν δεν υπάρχει καμία φορά ή η ρίζα του δένδρου είναι κενός δείκτης (`root==nullptr`), η μέθοδος επιστρέφει την τιμή **μηδέν** (0).

Η διαδικασία της δυαδικής αναζήτησης είναι **λογαριθμικής πολυπλοκότητας** $\Theta(\lceil \log(n + 1) \rceil)$, $n :=$ πλήθος κόμβων, και συνοψίζεται στην εξής μεθοδολογία:

Έστω ότι αναζητείται η συμβολοσειρά `word`. Εφόσον η ρίζα δεν είναι `nullptr`, συγκρίνεται ως προς τη σχέση διάταξης η `word` με τη ρίζα και αν είναι αλφαριθμητικά μεγαλύτερη, εξετάζεται ως προς τη σχέση διάταξης η `word` με το δεξί κόμβο-παιδί της ρίζας, διαφορετικά με το αριστερό. Η διαδικασία επαναλαμβάνεται καθ' ομοίως, μέχρι να προσεγγιστεί κόμβος p για τον οποίο ισχύει: $p->data!=word$, και ο p δεν έχει παιδιά, οπότε η αναζήτηση επιστρέφει **0**, ή να προσεγγιστεί κόμβος k , τέτοιος ώστε $k->data==word$, οπότε η μέθοδος επιστρέφει την τιμή $k->num$, δηλαδή το **πλήθος εμφανίσεων** της συμβολοσειράς `word` στο κείμενο.

Διάσχιση Δυαδικού Δένδρου

Για την καθολική προσέλαση/διάσχιση των κόμβων ενός αντικειμένου τύπου `BSTree`, υλοποιήθηκαν τρεις μέθοδοι που πραγματοποιούν:

- › **προδιατεταγμένη** διάσχιση (preOrder()) : ρίζα → αριστερός κόμβος → δεξιός κόμβος
- › **ενδοδιατεταγμένη** διάσχιση (inOrder()) : αριστερός κόμβος → ρίζα → δεξιός κόμβος
- › **μεταδιατεταγμένη** διάσχιση (postOrder()) : αριστερός κόμβος → δεξιός κόμβος → ρίζα

Οι μέθοδοι αυτές λειτουργούν αναδρομικά και εμφανίζουν πρώτα τα κατάλληλα υποδένδρα ή τη ρίζα και έπειτα τα αντιδιαμετρικά τους. Η κάθε μια καλεί την αντίστοιχη **προστατευμένη μέθοδο** (τύπος Διάσχισης (BTNode*)) προκειμένου να ξεκινήσει η αναδρομική διαδικασία, αρχίζοντας με τη ρίζα (root) ως αρχικό όρισμα. Το όρισμα αυτό ανανεώνεται σε κάθε αναδρομική κλήση με τον κόμβο-ρίζα του κατάλληλου υποδένδρου (συναρτήσει της διάσχισης που εκτελείται).

3.4 Δένδρο Αναζήτησης AVL

Η δομή του δυαδικού δένδρου αναζήτησης AVL αφορά στα αρχεία AVLtree.h και AVLtree.cpp. Το αρχείο βιβλιοθήκης (.h) περιλαμβάνει τη (δημόσια) κλάση AVLTree, καθώς και το struct avlNode, το οποίο παριστάνει κόμβους ενός δυαδικού δένδρου. Σημειώνεται ότι η εν λόγω κλάση θα μπορούσε να υλοποιηθεί ως **υποκλάση** (/παιδί-κλάση) της δομής BSTree, καθώς διαφέρουν μόνο στη διαδικασία εισαγωγής στοιχείων. Ωστόσο, αποφεύχθηκε αυτή τη τακτική κληρονομικότητας για λόγους **πλήρους ανεξαρτησίας** κάθε δομής δεδομένων.

```
class AVLTree {
private:
    avlNode *root;
    long height(avlNode *);
    long difference(avlNode *);
    avlNode * rr_rotate(avlNode *);
    avlNode * ll_rotate(avlNode *);
    avlNode * lr_rotate(avlNode*);
    avlNode * rl_rotate(avlNode *);
    avlNode * balance(avlNode *);
    avlNode * insert(avlNode*, const string &);
    void inOrder(avlNode *);
    void preOrder(avlNode *);
    void postOrder(avlNode* );
    void deleteAVL(avlNode* );
    void deleteAVL();
public:
    AVLTree();
    ~AVLTree();
    int search(const string &);
    void insert(const string &);
    bool deleteWord(const string &);
    void inOrder();
    void preOrder();
    void postOrder();
    long getHeight();
};

struct avlNode {
    int num;
    string data;
    avlNode *left;
    avlNode *right;
    avlNode(){
        left = nullptr;
        right = nullptr;
    }
    avlNode(const string &word) {
        data = word;
        left = nullptr;
        right = nullptr;
        num = 1;
    }
};
```

Κώδικας 5. Η κλάση AVLTree.

Κώδικας 4. Η δομή κόμβου δένδρου AVL avlNode.

Περιγραφή της δομής avlNode

Για τη σύνθεση του δένδρου AVL, δημιουργήθηκε μια δομή (struct) για την **αναπαράσταση** κάθε **κόμβου**-στοιχείου του. Η χρησιμότητα και η λειτουργία της δομής αυτής είναι **όμοια με** εκείνη του **BTNode**, με το κάθε αντικείμενο να αποτελεί έναν κόμβο του δένδρου, περιέχοντας μια (πρωτότυπη) λέξη (data) του κειμένου του αρχείου εισόδου και το πλήθος εμφανίσεών της σ' αυτό (num).

Οι μέθοδοι **εισαγωγής** [insert(const string &), insert(avlNode *, const string &)] και **διαγραφής** [deleteWord(const string &)] λέξης από τη δομή έχουν παρόμοια λειτουργία με τις αντίστοιχες μεθόδους της κλάσης BSTree. Η διαφοροποίησή τους εντοπίζεται σε ένα επιπλέον κομμάτι κώδικα που έχει προστεθεί, το οποίο διατηρεί τη δενδρική δομή **ισοζυγισμένη**, καθώς εισάγονται και διαγράφονται στοιχεία. Πρακτικά, οι δύο λειτουργίες υλοποιήθηκαν κατά τα άλλα πανομοιότυπα, γι' αυτό δε γίνεται εκτενέστερη αναφορά.

Μετά από κάθε **εισαγωγή** ενός νέου κόμβου στη δενδρική δομή **ή διαγραφή** ενός υπάρχοντος από αυτή, ελέγχεται, μέσω των μεθόδων balance(avlNode *) και difference(avlNode *), η **υψομετρική διαφορά** h των υποδένδρων της, ώστε αυτή να διατηρείται πάντοτε στο διάστημα $|h| \leq 1$. Αν, λοιπόν, παραβιάζεται η **αρχή σύνθεσης του AVL ΔΔΑ** (: **μέγιστη υψομετρική διαφορά υποδένδρων το 1**), τότε η δομή πρέπει να **ισοσταθμιστεί**, οπότε εκτελούνται οι κατάλληλες **περιστροφές** των κόμβων των «προβληματικών» υποδένδρων, προκειμένου να αποκατασταθεί η υψομετρική διαφορά τους με τα υπόλοιπα φύλλα στα επιτρεπτά πλαίσια.

Οι περιστροφές αυτές πραγματοποιούνται με τις (προστατευμένες) μεθόδους: rr_rotate(avlNode *), ll_rotate(avlNode *), lr_rotate(avlNode*) και rl_rotate(avlNode *).

Αν ο **παράγοντας υψομετρικής διαφοράς** που προκύπτει από τη μέθοδο difference, δηλαδή η διαφορά των υψών των δύο υποδένδρων ενός κόμβου είναι > 1 , τότε:

- › είτε πρέπει να εκτελεστεί **left-left** περιστροφή,
- › είτε **left-right** περιστροφή.

Για να διακριθεί η κατάλληλη εξ αυτών, ελέγχεται η υψομετρική διαφορά των υποδένδρων του αριστερού υποδένδρου του τρέχοντος κόμβου (αναδρομική κλήση της difference) και αν είναι ≥ 0 , καλείται η **ll_rotate(avlNode *)**, διαφορετικά η **rr_rotate(avlNode *)**.

Αν ο **παράγοντας υψομετρικής διαφοράς** είναι < -1 , τότε:

- › είτε πρέπει να εκτελεστεί **right-right** περιστροφή,
- › είτε **right-left** περιστροφή.

Για να διακριθεί η κατάλληλη εξ αυτών, ελέγχεται η υψομετρική διαφορά των υποδένδρων του αριστερού υποδένδρου του τρέχοντος κόμβου (αναδρομική κλήση της difference) και αν είναι ≤ 0 , καλείται η **rr_rotate(avlNode *)**, διαφορετικά η **rl_rotate(avlNode *)**.

Αναζήτηση στοιχείων και Διάσκιση Δένδρου

Η διαδικασία της (**δυαδικής αναζήτησης** εντός της AVL δενδρικής δομής πραγματοποιείται με τη μέθοδο search(const string &), η οποία έχει υλοποιηθεί πανομοιότυπα με την αντίστοιχη του απλού ΔΔΑ.

Αναλογικά, οι τρεις μέθοδοι **διάσκισης** [postOrder(), preOrder(), inOrder()] του δένδρου είναι πανομοιότυπα υλοποιημένοι με τις αντίστοιχες της ίδιας κλάσης (BSTree).

3.5 Πίνακας Κατακερματισμού με ανοιχτή διεύθυνση

Η δομή του πίνακα κατακερματισμού αφορά στα αρχεία hashTable.h και hashTable.cpp. Το Header αρχείο περιλαμβάνει τη κλάση hashTable και την ιδιωτική κλάση Cell, η οποία περιλαμβάνει τα δεδομένα ενός στοιχείου της δομής (λέξη και εμφανίσεις).

```
class hashTable {  
private:  
    class Cell {  
        public:  
            string word;  
            int occurrences = 0;  
    };  
    long size = 256;  
    Cell *table = new Cell[size];
```

```

long occupied = 0;
long expand_threshold = floor(sqrt(size));
void update_threshold(){
    expand_threshold = floor(sqrt(size));
}
long stringToHash(const string word, long max) const;
void expandAndRehash();
public:
    bool insertUnique(string word, int occurrences);
    bool insert(string word);
    int search(string word);
    long getSize(){return occupied;}
};

```

Κώδικας 6. Η κλάση hashTable.

Περιγραφή της δομής hashTable

Για την απλοποίηση της σύνθεσης της δομής του πίνακα κατακερματισμού κρίθηκε απαραίτητη η κατασκευή μιας ιδιωτικής εμβέλειας κλάσης (Cell) για την αναπαράσταση του κάθε κόμβου/στοιχείου του.

Ένα στοιχείο τύπου Cell περιέχει μια μεταβλητή string word που περιλαμβάνει την αλφαριθμητική αναπαράσταση μιας λέξης και την μεταβλητή int occurrences που αρχικοποιείται 0 και περιλαμβάνει τις εμφανίσεις αυτής της λέξης.

Η δομή hashTable λειτουργεί μετατρέποντας την εισακτέα λέξη μέσω μιας συνάρτησης κατακερματισμού σε αριθμό κλειδί ο οποίος ανήκει σε ένα συγκεκριμένο εύρος τιμών και την καταχωρεί στη θέση ενός πίνακα που αντιστοιχεί σε αυτό το κλειδί.

Συνάρτηση Κατακερματισμού

Βασικός στόχος της συνάρτησης κατακερματισμού είναι να μετατρέπει λέξεις, περασμένες ως ορίσματα σε φυσικούς αριθμούς ενός εύρους. Επειδή το εύρος των αριθμών ενδέχεται να είναι μικρότερο του εύρους των λέξεων, είναι σημαντικό η συνάρτηση να προκαλεί το λιγότερο δυνατές συγκρούσεις.

Η λειτουργία αυτή επιτυγχάνεται με τον ακόλουθο βρόχο:

```

for (char i : word) {
    hash_val = (hash_val + (i + 1 - 'a') * seed) % m;
    seed = (seed * prime) % m;
}
return hash_val;

```

Κώδικας 7. Βρόχος κατακερματισμού.

Γίνονται η επαναλήψεις, όπου η το μέγεθος της λέξης και σε κάθε επανάληψη αποθηκεύεται ένα γράμμα στην μεταβλητή i. Στην συνέχεια το γράμμα μετατρέπεται στην αριθμητική του αναπαράσταση στην αλφαβήτα (px. A = 0, B = 1, ..., Z = 26), πολλαπλασάζεται με ένα πολλαπλάσιο πρώτου αριθμού και διαιρείται με m, όπου m το εύρος αριθμών που παράγονται.

Εισαγωγή στοιχείων

Παρέχονται δύο συναρτήσεις για την εισαγωγή στοιχείων στη δομή hashTable.

- `insert(const string& word)`: Δέχεται σαν όρισμα μια μεταβλητή τύπου string και υπολογίζει το κλειδί της μέσω της συνάρτησης κατακερματισμού. Ύστερα προσπελαύνεται η θέση του πίνακα στοιχείων που αντιστοιχεί στο κλειδί και ελέγχει αν το κελί είναι άδειο (περιέχει μη ορισμένη λέξη και εμφανίσεις 0), στην οποία

- περίπτωση εισάγει την λέξη, θέτει τις εμφανίσεις στο 1 και επιστρέφει **True** που σηματοδοτεί την επιτυχή εισαγωγή του στοιχείου. Σε περίπτωση που το κελί δεν είναι άδειο, ελέγχει αν περιέχει την λέξη και αυξάνει τις εμφανίσεις της κατά μια μονάδα και επιστρέφει **False** καθώς δεν προστέθηκε το στοιχείο. Εφόσον δεν είναι κενό το κελί και δεν περιέχει ήδη την λέξη, επαναλαμβάνεται όλη η διαδικασία για την επόμενη θέση (αν βρεθεί το τέλος του πίνακα, προσπελαύνεται το πρώτο κελί). Στο τέλος της επανάληψης, ελέγχεται αν χρειάζεται να επεκταθεί ο πίνακας.
- `insertUnique(const string& word, int occurrences)`: Η `insertUnique` λειτουργεί κατά τον ίδιο τρόπο με την `insert`, αλλά αντί να προσαυξάνει τις εμφανίσεις μιας λέξης κατά μια μονάδα, θέτει τη τιμή που λαμβάνει σαν όρισμα `occurrences`.

Αναζήτηση στοιχείων

Η αναζήτηση στοιχείων του πίνακα γίνεται μέσω της συνάρτησης `search(const string& word)`, η οποία δέχεται σαν όρισμα μια μεταβλητή τύπου `string` και υπολογίζει το κλειδί της μέσω της συνάρτησης κατακερματισμού. Ύστερα προσπελαύνεται η θέση του πίνακα στοιχείων που αντιστοιχεί στο κλειδί και ελέγχει αν το κελί είναι άδειο (περιέχει μη ορισμένη λέξη και εμφανίσεις 0), στην οποία περίπτωση επιστρέφει 0, για να σηματοδοτήσει ότι η λέξη δεν υπάρχει στον πίνακα. Σε περίπτωση που το κελί δεν είναι άδειο, ελέγχει αν περιέχει την λέξη και αν υπάρχει επιστρέφει τις καταγεγραμμένες εμφανίσεις. Εφόσον δεν είναι κενό το κελί και δεν περιέχει ήδη την λέξη, επαναλαμβάνεται όλη η διαδικασία για την επόμενη θέση μέχρι να βρεθεί η λέξη ή κενό κελί. (Λόγω της συνθήκης επέκτασης του πίνακα, δεν είναι δυνατόν να προκύψει ατέρμον βρόχος, διότι πάντα θα υπάρχουν κενά κελιά).

Επέκταση πίνακα

Η επέκταση του πίνακα γίνεται κάθε φορά που τα **μη κατειλημμένα** κελιά του είναι λιγότερα από την τετραγωνική ρίζα του συνολικού αριθμού των κελιών.

Η επέκταση επιτυγχάνεται με τη κλήση της συνάρτησης `expandAndRehash()`.

Η συνάρτηση δημιουργεί ένα αντίγραφο του πίνακα και διπλασιάζει την θέση μνήμης που ήταν καταχωρημένη στον αρχικό δείκτη του πίνακα καθώς και τη μεταβλητή `size`. Ύστερα μέσω επανάληψης γίνεται εισαγωγή του κάθε στοιχείου του παλιού πίνακα στον καινούργιο με τη χρήση της συνάρτησης `insertUnique`, υπολογίζοντας παράλληλα καινούργια κλειδιά και τέλος διαγράφοντας το παλαιό αντίγραφο του πίνακα.

4 Αξιολόγηση Αποτελεσμάτων Εκτέλεσης

Για τον έλεγχο της λειτουργίας των πέντε παραπάνω δομών υλοποιήθηκε η βασική συνάρτηση `main`, στην οποία κατασκευάζονται από ένα αντικείμενο της κάθε κλάσης και εκτελείται η παράλληλη εισαγωγή των λέξεων του αρχείου εισόδου (`.txt`), κατά τη σειριακή προσπέλαση του περιεχομένου του. Η εισαγωγή σε κάθε αντικείμενο γίνεται με την κατάλληλη συνάρτηση εισαγωγής της κάθε κλάσης. Έπειτα, εκτελείται η αναζήτηση Q τυχαίων λέξεων του αρχείου μέσα στις δομές (με τις αντίστοιχες συναρτήσεις κάθε κλάσης) και επιστρέφεται το πλήθος εμφανίσεων καθεμιάς, όπως αυτό αποθηκεύεται στις δομές. Τέλος, εκτελείται η διαγραφή μιας λέξης (της ίδιας) από κάθε δομή, θεωρώντας ότι απαλείφονται όλες οι εμφανίσεις της -δηλαδή, το πλήθος εμφανίσεων μπορεί να μειωθεί και η λέξη «αποκόπτεται» απ' την (κάθε) δομή.

Με τη χρήση της βιβλιοθήκης `chrono` υπολογίστηκε ο χρόνος που απαιτήθηκε από κάθε δομή για την **εισαγωγή όλων των πρωτότυπων λέξεων** του κειμένου εισόδου και την **αναζήτηση** διαφόρων συνόλων Q τυχαίων λέξεων του αρχείου εισόδου εντός των δομών, όπου Q σύνολο τέτοιο, ώστε $|Q| \cdot 10^{-3} \in \{1, 3, 5, 10, 15\}$. Οι αναζητήσεις πραγματοποιήθηκαν σε δύο αρχεία με διαφορετικά μεγέθη, προκειμένου να διαπιστωθεί η ισχύς της θεωρίας πολυπλοκότητας των υλοποιημένων αλγορίθμων αναζήτησης (Big-O notations). Τα αποτελέσματα που καταγράφηκαν παρατίθενται παρακάτω:

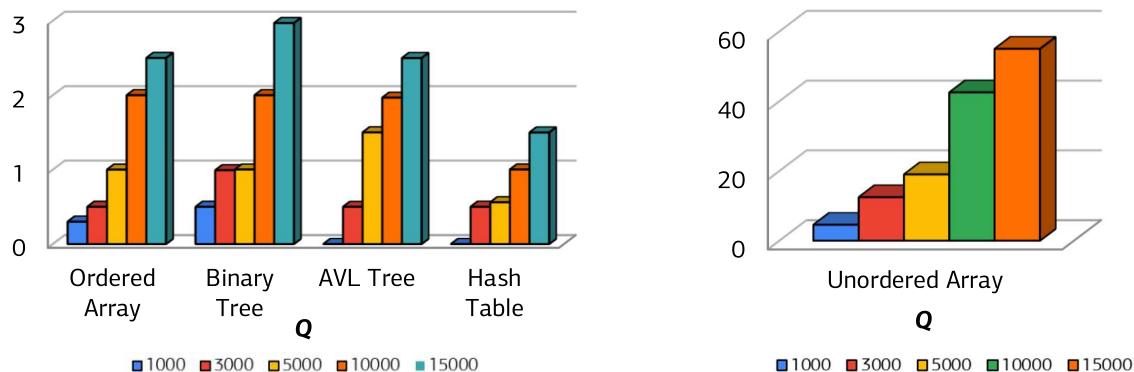
Δομές	Κατασκευή Δομών	
	Μικρό αρχείο	Μεγάλο αρχείο
Unordered Array	1.2340	!†
Ordered Array	3.4821	!
Binary Search Tree	0.1425	169.444
AVL BST	32.8285	!
Hash Table	0.1364	185.6310

Πίνακας 1. Χρόνος εισαγωγής σε seconds (s) όλων των λέξεων σε κάθε δομή.

Παρατηρούμε ότι οι δομές **Ordered Array** και **AVL Binary Search Tree** απαιτούν συγκριτικά περισσότερο χρόνο, αφού κατά την εισαγωγή των λέξεων εκτελούν πράξεις (operations) υψηλής πολυπλοκότητας (ταξινόμηση – περιστροφή, αντίστοιχα).

Δομές	1000	3000	5000	10000	15000
Unordered Array	4.5525	12.4644	19.0026	42.5122	54.9987
Ordered Array	0.3003	0.4999	0.9988	1.9993	2.4995
Binary Search Tree	0.4996	0.9908	0.9999	1.9996	2.9684
AVL BST	0.0000	0.4993	1.5000	1.9656	2.4982
Hash Table	0.0000 †	0.5005	0.5600	1.0009	1.4987

Πίνακας 2. Χρόνος αναζήτησης σε milliseconds (ms) Q λέξεων σε κάθε δομή στο μικρό αρχείο κειμένου (βλ. Σχήμα 5).



Σχήμα 5a. Αντιπαραβολή χρονικών απαιτήσεων των δομών Ordered Array, Binary Search Tree, AVL BST και Hash Table.

Σχήμα 5b. Διάκριση του Unordered Array για λόγους διαφορετικής κλίμακας. Οι χρονικές απαιτήσεις της εν λόγω δομής είναι μακράν υψηλότερες.

Δομές	1000	3000	5000	10000	15000
Binary Search Tree	0.0000	1.0016	1.4925	2.9909	5.9997
Hash Table	0.0000	0.5017	0.9991	1.4938	2.5317

Πίνακας 3. Χρόνος αναζήτησης σε milliseconds (ms) Q λέξεων σε κάθε δομή στο μεγάλο αρχείο κειμένου.

†! : Δε μπορούσε να γίνει καταγραφή των συγκεκριμένων χρονικών απαιτήσεων, καθώς οι εν λόγω δομές εκτελούν χρονοβόρες διαδικασίες, μη μετρήσιμες σε φυσιολογικά χρονικά πλαίσια, ιδιαίτερα, όταν αποκτούν αυξημένο μέγεθος (βλ. μεγάλο αρχείο).

‡ 0.000 : Οι μηδενικές μετρήσεις οφείλονται σε έλλειψη ακρίβειας της βιβλιοθήκης chrono.

- Με τη **γραμμική αύξηση** του $|Q|$ παρατηρείται ότι αυξάνεται **γραμμικά και ο χρόνος αναζήτησης** των λέξεων (ενν. στο ίδιο αρχείο – n αμετάβλιπτο, βλ. Πίνακας 2).
- Συγκρίνοντας τις χρονικές/υπολογιστικές απαιτήσεις του προγράμματος για την αναζήτηση των διαφόρων συνόλων Q λέξεων μεταξύ των δύο αρχείων, παρατηρούμε ότι η αύξηση του μεγέθους n των αντίστοιχων δομών επαληθεύει τη θεωρία.

Αναλυτικότερα, το **Δυαδικό Δένδρο Αναζήτησης** πράγματι απαιτεί χρόνο $O(\log n)$ για να εντοπίσει $|Q| = k$ λέξεις εντός της δομής, αφού καθώς το n αυξήθηκε -από το μικρό στο μεγάλο αρχείο-, ο απαιτούμενος χρόνος ακολούθησε λογαριθμική καμπύλη. Επιπλέον, είναι η **δεύτερη καλύτερη** δομή από πλευράς **κατασκευής** και **τρίτη** από **αναζήτησης**, όπως αποδεικνύεται και στα δύο αρχεία, καθώς το Q μεταβάλλεται.

Κατ' αναλογία, ο **Πίνακας Κατακερματισμού Ανοικτής Διεύθυνσης** απαιτεί (περίπου) σταθερό χρόνο $O(1)$ για να εκτελέσει τις αντίστοιχες διαδικασίες αναζήτησης, πράγμα το οποίο επαληθεύεται μετά την αύξηση του $|Q|$. Ακόμη σε μεταξύ αναζητήσεων εντός του ίδιου αρχείου, αναδεικνύεται στην **αποτελεσματικότερη** (ταχύτερη) **δομή**, τόσο από άποψη **κατασκευής**, όσο και από άποψη **αναζήτησης**.

- Ο **Αταξινόμητος Πίνακας** απαιτεί συγκριτικά με τις άλλες δομές πολύ περισσότερο χρόνο για την αναζήτηση των λέξεων. Αυτό, εξάλλου, ήταν αναμενόμενο υπό το πρίσμα της **σειριακής αναζήτησης** που εκτελεί, με πολυπλοκότητα ύψους $O(n)$.
- Ο **Ταξινομημένος Πίνακας** χρειάζεται περισσότερο χρόνο από τον αταξινόμητο για να κατασκευαστεί, αλλά η **δυαδική αναζήτηση** που εκτελεί είναι πολύ περισσότερο αποδοτική, αφού με $O(\log n)$ η διαφορά ακόμη και στο πλήθος των αναζητούμενων λέξεων -θεωρητικά και με τη μεταβολή του n (δεν ήταν δυνατό να ελεγχθεί λόγω αργής κατασκευής)- δεν μεταβάλλει σε σημαντικά πλαίσια τις χρονικές απαιτήσεις της αναζήτησης.
- Το **Δένδρο AVL** αποδεικνύεται η υπολογιστικά «βαρύτερη» δομή από άποψη κατασκευής, αφού εκτελεί πολύ συχνά συναρτήσεις ελέγχου υψομετρικής διαφοράς και περιστροφής, προκειμένου να παραμένει ζυγισμένο. Ωστόσο, η **αναζήτηση** δικαιώνει τη δομή, αφού η ταξινομημένη διάταξη που διατηρεί στα φύλλα του δένδρου την καθιστά ταχύτατη.

Κατά την καταγραφή των αποτελεσμάτων παρατηρούνται διάφορα **σφάλματα** -δηλαδή, μετρήσεις που αποκλίνουν από τις αναμενόμενες τιμές- τα οποία θα μπορούσαν να θεωρηθούν αμελητέα ή αποδεκτά, αφού στη διάρκεια των μετρήσεων το λειτουργικό σύστημα εκτελεί παράλληλες διεργασίες παρασκηνίου, οι οποίες επηρεάζουν τη λειτουργία του προγράμματος (μεταβολές στη μνήμη RAM κλπ.), με αποτέλεσμα η κάθε διαφορετική εκτέλεση (Run) να αποδίδει αποτελέσματα με αποκλίνουσες τιμές ορισμένου ανεκτού ορίου. Για τον λόγο αυτό, το πρόγραμμα εκτελέστηκε αρκετές φορές (IDEs: CLion, Replit.com – OSs: Windows, Linux) και καταγράφηκαν οι δειγματικοί μέσοι όροι χρονικών απαιτήσεων κάθε δομής.