



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	XXXX	Μπαλακτσής Χρήστος
	XXXX	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. Επιβλέπων διδάσκων της εργασίας είναι ο κ.κ. Απόστολος Παπαδόπουλος. 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**¹ 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 $5000 \cdot k$, $k = 1, \dots, \theta$ σεις του. Για να πραγματοποιηθεί αυτό, επειδή χρησιμοποιούνται συνεχόμενες θέσεις

¹ 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 χρησιμοποιημένου 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:

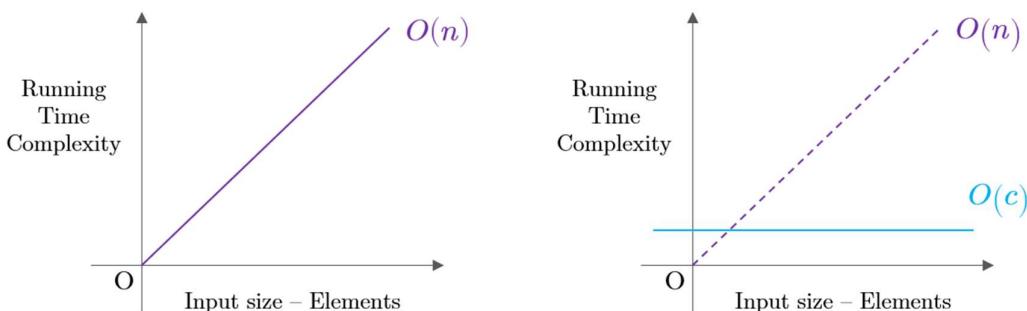
```
› int getSize() const;           // πρόσβαση στο τρέχον συνολικό μέγεθος του πίνακα
› int getNum(long) const;         // πρόσβαση στο num[<όρισμα>]
› string getData(long) const;     // πρόσβαση στο data[<όρισμα>]
› void setNum(long, int);         // ορισμός της τιμής num[<όρισμα>]
```

Εισαγωγή στοιχείων

Η πρώτη, μαζική εισαγωγή των λέξεων στη δομή γίνεται με τη μέθοδο `insertUnique(const string &, int)`, η οποία προϋποθέτει έλεγχο με άλλο-εξωτερικό τρόπο των διπλότυπων όρων, δέχεται ως όρισμα την πρωτοεμφανιζόμενη νέα λέξη και έναν ενδεικτικό ακέραιο αριθμό εμφανίσεων, ο οποίος, όμως, ενημερώνεται παρακάτω στο ορθό, μετά την εισαγωγή όλων των πρωτότυπων λέξεων, και, συνεπώς, το όρισμα αυτό είναι αδιάφορο και παίρνει πάντα ως ενδεικτική τιμή το 1.

Ο λόγος που επιλέχθηκε αυτήν τη μέθοδο εισαγωγής έναντι της άμεσης εισαγωγής με αναζήτηση και εισαγωγή της λέξης αποκλειστικά μέσω της δομής του αταξινόμητου πίνακα συνοψίζεται στον χρόνο που απαιτεί η διαδικασία της σειριακής αναζήτησης ενός κλειδιού (λέξης) για ένα κείμενο 2.2 GB στην αταξινόμητη δομή.

Η **εξασφάλιση της μοναδικότητας** της κάθε λέξης, λοιπόν, γίνεται με την αναζήτηση της λέξης στον πίνακα κατακερματισμού, αφού εκεί το κόστος αναζήτησης είναι μόλις $O(c)$, $c \in \mathbb{R}$, έναντι του $O(n)$, $n :=$ πλήθος στοιχείων της δομής, που αφορά τον αταξινόμητο πίνακα. Συνεπώς, αρκεί να εξασφαλισθεί ότι η τρέχουσα προσπελαύνουσα λέξη του αρχείου **δεν περιέχεται** στον πίνακα κατακερματισμού, ώστε να εισαχθεί και στις δύο προαναφερθείσες δομές. Σημειώνεται ότι, όταν προσπελαύνεται μη-πρωτότυπη λέξη, ενημερώνεται μεν το πλήθος εμφάνισης της συγκεκριμένης λέξης στον πίνακα κατακερματισμού, αλλά ο αταξινόμητος πίνακας παραμένει αμετάβλητος. Η ενημέρωση του πλήθους εμφάνισης κάθε λέξης σε αυτόν απαιτεί την αντιγραφή της συγκεκριμένης πληροφορίας από τη δομή του πίνακα κατακερματισμού, μετά το τέλος της εισαγωγής νέων λέξεων.



Σχήμα 1. Μείωση του κόστους των πράξεων (operations) από γραμμικό σε σταθερό με την αποφυγή της σειριακής αναζήτησης και αύξηση της απόδοσης του προγράμματος.

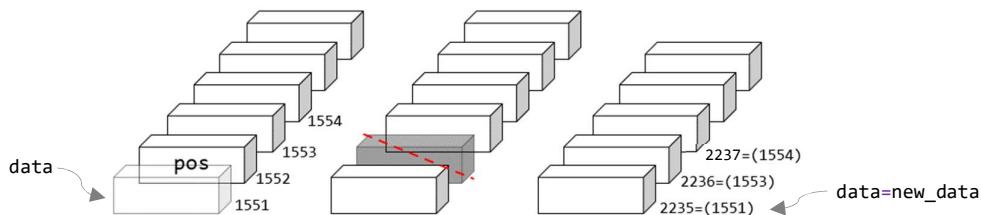
Πέρα, όμως, από τη μαζική εισαγωγή των λέξεων του κειμένου, υλοποιήθηκε σχετική μέθοδος για τη **μεμονωμένη εισαγωγή** μια λέξης στη δομή. Ο λόγος γίνεται για τη μέθοδος `insert(const string &)`, η οποία δέχεται μια συμβολοσειρά (λέξη), την αναζητεί σειριακά (χρησιμοποιώντας την ιδιωτική μέθοδο (σ.σ. συνάρτηση) `Search_help(const string &, long &)`) εντός της δομής του αταξινόμπου 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 `insert_Unique`, που αναφέρθηκε παραπάνω, 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 περιεγράφη παραπάνω).

Διαγραφή στοιχείου

Η διαγραφή μιας λέξης από τη δομή του αταξινόμπου πίνακα συνεπάγεται τη διαγραφή/παράλειψη των κόμβων/στοιχείων του αταξινόμπου 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 –χρησιμοποιώντας 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).



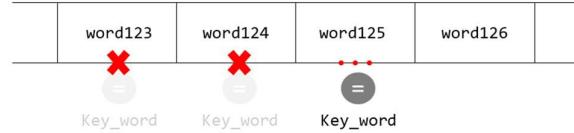
Σχήμα 2. Διαγραφή του στοιχείου της θέσης `pos` σε δυναμικό πίνακα με συμβατική αναπαράσταση. Η τρίτη συστοιχία αντιστοιχεί σε άλλο σημείο της μνήμης, όπου αντιγράφηκαν τα επιθυμητά κελιά που αντιστοιχούσαν στην προηγούμενη μορφή της δομής.

Αναζήτηση στοιχείου

Η αναζήτηση μιας συγκεκριμένης λέξης που **εν δυνάμει** υπάρχει στο κείμενο και κατ' επέκταση στη δομή του αταξινόμπου 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 αταξινόμπους 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. Στην 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.



Σχήμα 3. Σειριακή σύγκριση κάθε μια λέξης του πίνακα `data` με τη λέξη που δόθηκε ως όρισμα για αναζήτηση.

3.2 Ταξινομημένος πίνακας

Η δομή του ταξινομημένου πίνακα αποτελείται από τα αρχεία `orderedArray.h` και `orderedArray.cpp`. Το Header αρχείο περιλαμβάνει την κλάση `orderedArray`.

```
class orderedArray {
private:
    string *data;
    int *num;
    long size;
    bool binSearch(const string &word, long &pos);
    void quicksort(long start, long end);
    void swap(int &a, int &b);
    void swap(string &a, string &b);
public:
    orderedArray();
    void copyFromUnordered(string *newData, int *newNum,
                           long arraySize);
    void insert(const string& word);
    void insertUnique(const string& word, int occurrences);
    int search(const string& word);
    void remove(const string& word);
    int 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 στον πίνακα `num`. Σε περίπτωση μη εύρεσής της, θα εκτελέσει μια λειτουργία όμοια της `insertion-sort`, όπου θα αναζητηθεί η θέση κ στην οποία θα άνηκε η λέξη ώστε να είναι ταξινομημένος ο πίνακας `data` και θα γίνει ολίσθιση των στοιχείων σε υψηλότερες θέσεις κατά μια μονάδα, ενώ στην κ θα 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, που 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 τις εσωτερικές δομές `data` και `num` τους 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 της `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();  
};
```



```
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 3. Η κλάση `BSTree`.

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(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 δεξιό παιδί του προηγούμενου κόμβου.

Το **κριτήριο** καθορισμού της **κατεύθυνσης** (δεξιά-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.

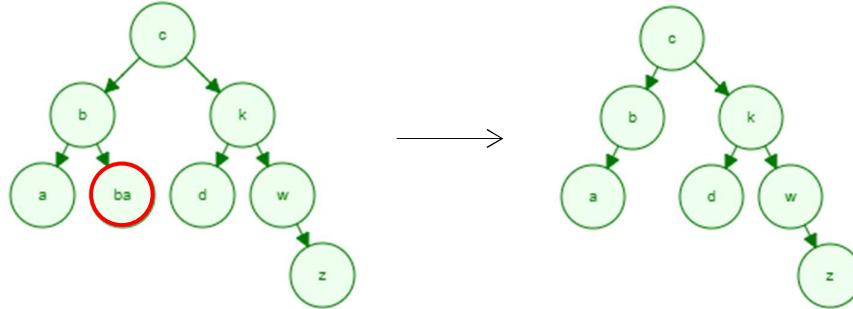
Πρακτικά, η διαδικασία που περιγράφεται παραπάνω υλοποιείται με τη βοήθεια της προστατευμένης μεθόδου `insert(BTNode*, const string &)`. Συγκεκριμένα, η κλήση της `insert(const string &)` ενεργοποιεί την προηγούμενη ομώνυμη συνάρτηση με ορίσματα τη ρίζα `root` και τη συμβολοσειρά που δόθηκε από την `main`. 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 στοιχείου

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 εμφανίσεών της εντός του κειμένου. Η

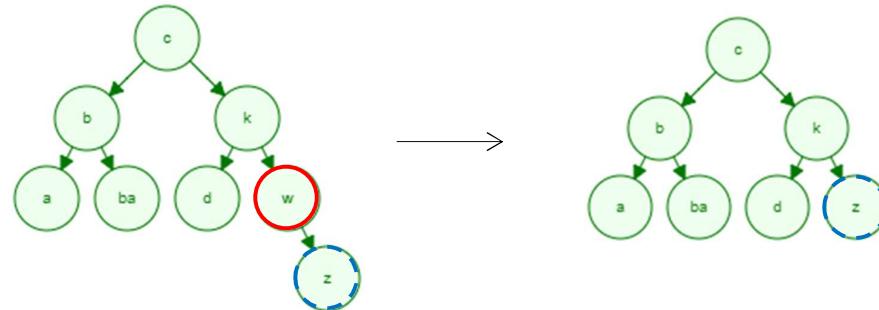
διαδικασία αυτή πραγματοποιείται μέσω της μεθόδου `deleteWord(const string &)` που εφαρμόζεται επί της κλάσης `BSTree`. Η μέθοδος δέχεται ως όρισμα μια συμβολοσειρά και ξεκινάει τη δεικτοδοτούμενη αναζήτησή της εντός της δενδρικής δομής και:

- αν **δεν υπάρχει**, επιστρέφει ***False*** (Boolean Value).
- αν **υπάρχει** σε έναν κόμβο, ύστοι 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 .



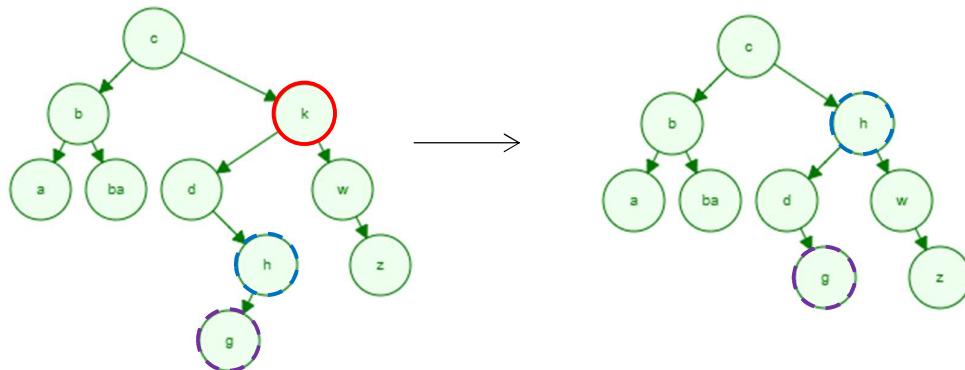
Σχήμα 4. Διαγραφή του κόμβου-φύλλου.

- » αν έχει **ένα παιδί**, ύστοι 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 5. 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 6. 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 **επιστέψει 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 **λογαριθμική 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 προσπέλαση/διάσχιση των κόμβων ενός αντικειμένου τύπου `BSTree`, υλοποιήθηκαν τρεις μέθοδοι που πραγματοποιούν:

- **προδιατεταγμένη** διάσχιση (`preOrder()`) : $\text{ρίζα} \rightarrow \text{αριστερός κόμβος} \rightarrow \text{δεξιός κόμβος}$
- **ενδοδιατεταγμένη** διάσχιση (`inOrder()`) : $\text{αριστερός κόμβος} \rightarrow \text{ρίζα} \rightarrow \text{δεξιός κόμβος}$
- **μεταδιατεταγμένη** διάσχιση (`postOrder()`) : $\text{αριστερός κόμβος} \rightarrow \text{δεξιός κόμβος} \rightarrow \text{ρίζα}$

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*);  
  
    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;
```

```

    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();
};

}

```

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).

Εισαγωγή και Διαγραφή στοιχείων

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 Διάσχιση 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 **διάσχισης** [`postOrder()`, `preOrder()`, `inOrder()`] του δένδρου είναι πανομοιότυπα υλοποιημένοι με τις αντίστοιχες της ίδιας κλάσης (BSTree).

3.5 Πίνακας Κατακερματισμού με ανοιχτή διεύθυνση

Η δομή