Εργασία στις Δομές Δεδομένων

Νικόλαος Βογιατζής, Ολγκέρ Χότζα

Εαρινό εξάμηνο 2021

Περίληψη

Στις παρούσες διαφάνειες θα παρουσιαστούν μερικές δομές δεδομένων ως προς την λειτουργία και την υλοποίηση τους σε γλώσσα προγραμματισμού C++. Η κάθε δομή αξιολογείται με βάση τον χρόνο εκτέλεσης και τερματισμού κάποιον συγκεκριμένων ενεργειών/αλγορίθμων.

Περιεχόμενα

1 Εισαγωγή **1**

2 Αταξινόμητος πίνακας **2**
2.1 Μέθοδος εισαγωγής 3
2.2 Μέθοδος αναζήτησής 3
2.3 Μέθοδος διαγραφής 4
2.4 Σχολιασμός και χρόνοι εκτέλεσης 4

3 Ταξινομημένος πίνακας **6**
3.1 Μέθοδος εισαγωγής 6
3.2 Μέθοδος αναζήτησής 7
3.3 Μέθοδος ταξινόμησής 8
3.4 Σχολιασμός και χρόνοι εκτέλεσης 8

4 Απλό και δυαδικό δένδρο αναζήτησής τύπου AVL **10**
4.1 Μέθοδος εισαγωγής 10
4.2 Μέθοδος αναζήτησής 11
4.3 Μέθοδος διαγραφής 11
4.4 Σχολιασμός και χρόνοι εκτέλεσης 11

5 Πίνακας κατακερματισμού με ανοικτή διεύθυνση **12**
5.1 Μέθοδος εισαγωγής 13
5.2 Μέθοδος αναζήτησής 13
5.3 Σχολιασμός και χρόνοι εκτέλεσης 14

1 Εισαγωγή

Οι δομές που έχουμε υλοποιήσει και αξιολογήσει είναι οι παρακάτω πέντε:

1. αταξινόμητος πίνακας,
2. ταξινομημένος πίνακας,
3. απλό δυαδικό δέντρο αναζήτησης,
4. δυαδικό δένδρο αναζήτησης τύπου AVL και
5. πίνακας κατακερματισμού με ανοικτή διεύθυνση.

Για τις προαναφερθείσες δομές δεδομένων, με βάση το κείμενο από το Gutenberg Project (https://www.gutenberg.org/) που περιέχει γνωστά έργα παγκόσμιας λογοτεχνίας, έχουμε καταγράψει τους χρόνους ως προς την εισαγωγή και αναζήτηση.

2 Αταξινόμητος πίνακας

Περιγραφή

Η δομή ενός μη ταξινομημένου πίνακα είναι μια συλλογή στοιχείων όπου κάθε στοιχείο κατέχει μια σχετική θέση σε σχέση με τα άλλα.

Γενικά, ένας μη ταξινομημένος πίνακας είναι μια δομή δεδομένων που τα στοιχεία του είναι διατεταγμένα με βάση τη σειρά που εισήχθησαν στον πίνακα. Κάθε στοιχείο του πίνακα διαπερνάται με τη σειρά, έτσι για παράδειγμα η αναζήτηση γίνεται σειριακά έως ότου βρεθεί το επιθυμητό στοιχείο, διαφορετικά η σύγκριση συνεχίζεται μέχρι τέλους του πίνακα.

Θέση i στοιχείου	Περίπτωση	Συγκρίσεις	Πολυπλοκότητα
0	Καλύτερή	1	$O(1)$
$n-1$	Χειρότερή	n	$O(n)$

Πίνακας 2.1: Περιπτώσεις της σειριακής αναζήτησης

Για να μελετήσουμε θεωρητικά τις δυο περιπτώσεις ενός αλγορίθμου θα πρέπει να το δούμε υπό μαθηματικής άποψης. Έστω ότι n το σύνολο των στοιχείων και i η θέση του κάθε στοιχείου στον πίνακα. Η **καλύτερη περίπτωση** είναι το στοιχείο που αναζητούμε να βρίσκεται στη πρώτη θέση του πίνακα ($i = 0$). Αντίστοιχα, η **χειρότερη περίπτωση** θα ήταν αν το στοιχείο που αναζητούμε είναι το τελευταίο στοιχείο του πίνακα ($i = n - 1$) ή το στοιχείο που αναζητούμε να μην περιεχόταν στο πίνακα (βλ. Πίνακας 2.1).

Πριν ξεκινήσουμε να υλοποιούμε την εισαγωγή, αναζήτηση και διαγραφή του μη ταξινομημένου πίνακα (unorderedArray), θα πρέπει να ξεκινήσουμε την υλοποίηση της κλάσης.

```
#pragma once
#include <string>

using std::string;

class unorderedArray
{
private:
    struct Arr
    {
        string value;
        long long unsigned int times;
    };
    Arr* Array;
    long long unsigned int FirstFreeCell;
protected:
    bool ReallocateArray(int);
public:
    unorderedArray();
    ~unorderedArray();

    void Insert(string);
    bool Delete(string);
    bool Find(string, long long unsigned int&, long long unsigned int&);
    long long unsigned int getFirstFreeCell() { return FirstFreeCell; }

    string& operator[](unsigned int possition) { return Array[possition].value; }
};
```

Πρόγραμμα 2.1: Υλοποίηση της κλάσης για τον αταξινόμητο πίνακα

Όπως φαίνεται παραπάνω (Πρόγραμμα 2.1), κατασκευάζουμε εντός της κλάσης της UnorderedArray ένα struct το όποιο αποθηκεύει την λέξη μας καθώς και πόσες φορές αποθηκεύεται στον παρακάτω πίνακα.

Διατηρούμε την πρώτη ελεύθερη θέση σε μια μεταβλητή τύπου long long unsigned int για να βεβαιωθούμε πως θα φτάσουν τα bytes για την αποθήκευση του πλήθους λέξεων που θα αποθηκευτούν $+1$ θέση, όπου θα είναι η ελεύθερη θέση.

Στο δημόσιο τμήμα της κλάσης της αταξινόμητης τομής, υπάρχει ο κατασκευαστής, καταστροφέας και διάφορες συναρτήσεις τις οποίες θα μελετήσουμε παρακάτω.

2.1 Μέθοδος εισαγωγής

Ας υποθέσουμε ότι έχουμε έναν δυναμικό πίνακα με n θέσεις και i θα ονομάσουμε το κάθε στοιχείο του πίνακα με $i \in [0, n)$ και j το στοιχείο που θέλουμε να εισάγουμε στον πίνακα. Σε έναν μη ταξινομημένο πίνακα, η λειτουργία εισαγωγής είναι ταχύτερη σε σύγκριση με έναν ταξινομημένο πίνακα επειδή δεν χρειάζεται να ενδιαφερόμαστε για τη θέση στην οποία πρόκειται να τοποθετηθεί το στοιχείο.

Κάθε στοιχείο j θα τοποθετείτε στην τελευταία θέση του πίνακα. Για να λύσουμε το θέμα του πλήρη πίνακα, κάθε φορά θα δεσμεύουμε $n + 1$ θέσεις για κάθε j στοιχείο. Επομένως, ας δημιουργήσουμε δύο συναρτήσεις της κλάσης μας όπου η πρώτη συνάρτηση θα πραγματοποιεί την εισαγωγή στοιχείων στην τελευταία θέση του πίνακα και δεύτερων τη συνάρτηση που θα δεσμεύει νέα μνήμη.

Πρακτικά, στην δική μας υλοποίηση, λαμβάνοντας υπόψιν ότι θέλουμε να γνωρίζουμε και το πλήθος εμφανίσεων κάθε λέξης, χρησιμοποιούμε την συνάρτηση Find για να εντοπίσουμε αν η λέξη που θα εισάγουμε υπάρχει ή οχι. Σε περίπτωση που υπάρχει, τότε απλά αυξάνουμε κατά μια μονάδα την μεταβλητή εμφάνισης της λέξης. Σε διαφορετική περίπτωση, τότε δεσμεύουμε μια μονάδα περισσότερο χώρο (βλ. Πρόγραμμα 2.2 - Συνάρτηση ReallocateArray(int)) για την Array. Στη συνέχεια, αφού έχει δεσμευτεί χώρος, απλά εισάγουμε την λέξη στον πίνακα και θέτουμε ότι η λέξη είναι η πρώτη (δηλαδή έχει πλήθος εμφανίσεων μια φορές).

```
void unorderedArray::Insert(string word)
{
    long long unsigned int temp, times;
    if(Find(word, temp, times))
        Array[temp].times++;
    else
    {
        ReallocateArray(1);
        Array[First_Free_Element-1].value = word;
        Array[First_Free_Element-1].times = 1;
    }
}

bool unorderedArray::ReallocateArray(int n)
{
    Arr* temp = new (nothrow) Arr[First_Free_Element+n];

    if (temp == nullptr) return false;

    for (long long unsigned int element = 0; element < First_Free_Element; element++)
    {
        temp[element].value = Array[element].value;
        temp[element].times= Array[element].times;
    }

    delete[] Array;
    Array = temp;

    First_Free_Element++;
    return true;
}
```

Πρόγραμμα 2.2: Δημιουργία των συναρτήσεων Insert και RelocateArray του αταξινόμητου πίνακα

2.2 Μέθοδος αναζήτησής

Σε έναν μη ταξινομημένο πίνακα, η λειτουργία αναζήτησης θα εκτελεστεί με γραμμική διέλευση από το πρώτο στοιχείο $(i = 0)$ στο τελευταίο στοιχείο $(i = n - 1)$. Η διαδικασία αυτή ως προς την ταχύτητα εύρεσης ενός στοιχείου k (όπου k

το ενδιαφερόμενο στοιχείο), παρουσιάζει καθυστέρησή ανάλογα με τη θέση i που βρίσκεται το στοιχείο k.

Με βάση τον Πίνακα 2.1 παρατηρούμε ότι υπάρχει χειρότερη περίπτωση n συγκρίσεων σε περίπτωση που το k στοιχείο βρίσκεται στη θέση $n-1$ του πίνακα ή δεν περιέχεται. Έτσι λοιπόν, ο αλγόριθμος της σειριακής αναζήτησης έχει πολυπλοκότητα χειρότερης περίπτωσής $O(n)$.

Όπως φαίνεται στο Πρόγραμμα 2.3, κάθε φορά που καλείται η συνάρτηση, γίνεται έλεγχος σε περίπτωση που ο πίνακας είναι άδειος. Για κάθε περίπτωση που υπάρχουν έστω ένα ή περισσότερα στοιχεία/λέξεις μέσα στον πίνακα, τότε σειριακά επιτελείται η αναζήτηση. Αν το επιθυμητό στοιχείο βρεθεί, τότε, με αναφορά επιστρέφεται η θέση στην οποία βρέθηκε το στοιχείο, πόσες φορές βρέθηκε (που υπάρχει ήδη από την εισαγωγή) και τέλος επιστρέφεται ότι βρέθηκε. Για κάθε άλλη περίπτωση επιστρέφονται – με αναφορά – μηδενικές εμφανίσεις και false.

```
bool unorderedArray::Find(string object, int &temp, int &existance_times)
{
    if(First_Free_Element == 0)
        return false;

    for (int i = 0; i < First_Free_Element; i++)
    {
        if(Array[i].value==object)
        {
            temp = i;
            existance_times = Array[i].times;
            return true;
        }
    }

    existance_times = 0;
    return false;
}
```

Πρόγραμμα 2.3: Δημιουργία της συνάρτησής Find του αταξινόμητου πίνακα

2.3 Μέθοδος διαγραφής

Η διαγραφή στον αταξινόμητο πίνακα είναι χρονοβόρα καθώς γίνεται χρήση της συνάρτησης Find και διαπερνούμε όλα τα στοιχεία του πίνακα. Η μεθοδολογία που ακολουθούμε για να καλύψουμε τις κενές θέσεις είναι να αλλάζουμε την θέση του κάθε στοιχείου του πίνακα κατά μια θέση αρνητικά, με βάση τη θέση την οποία βρέθηκε το επιθυμητό στοιχείο που είναι προς διαγραφή.

Υπό αλγοριθμικής σκέψης, θα λέγαμε ότι, αν k το στοιχείο προς αναζήτηση και $P(k)$ η θέση στην οποία βρέθηκε, τότε όλα τα στοιχεία με θέσεις $\geq P(k)$ θα αντιμετατεθούν τις θέσεις τους κατά μείον μια θέση από την αρχική τους.

Υλοποιώντας την σκέψη μας με $C++$ κώδικα (βλ. Πρόγραμμα 2.4), αρχικά θα ελέγξουμε αν το στοιχείο που επιθυμούμε να διαγράψουμε υπάρχει. Αν δεν υπάρχει τότε τερματίζουμε την συνάρτηση και επιστρέφουμε false. Διαφορετικά, υλοποιούμε την παραπάνω σκέψη μας σε αλγόριθμο. Τέλος, μειώνουμε την τελευταία ελεύθερη θέση του πίνακα κατά μια μονάδα και επιστρέφουμε true.

2.4 Σχολιασμός και χρόνοι εκτέλεσης

Ο χρόνος εισαγωγής σε έναν αταξινόμητο πίνακα, σαφώς και είναι λιγότερος σε σχέση με την δομή του ταξινομημένου πίνακα. Κάθε στοιχείο μεταφορτώνεται στο τέλος του πίνακα, οπότε, κατ-ανάγκη, δεν την καθιστά, ως προς την εισαγωγή, χειρότερη δομή.

Δομή	Χρόνος Εισαγωγής	Χρόνος Αναζήτησης/10χιλ. λέξεις
Αταξινόμητος πίνακας	≈10.9451192 ώρες	≈33.1032167 λεπτά

Πίνακας 2.2: Χρόνοι εκτέλεσης εισαγωγής και αναζήτησης για τον αταξινόμητο πίνακα

Όσο για την αναζήτηση, επειδή κάθε στοιχείο αναζητείται σειριακά, δηλαδή συγκρίνουμε όλα τα προηγούμενα στοιχεία με το στοιχείο που ψάχνουμε, καθιστά την δομή αργή. Αν δώσουμε μια εκτίμηση για δέκα χιλιάδες λέξεις έχουμε τα παραπάνω χρονικά αποτελέσματα (Πίνακας 2.2).

```
bool unorderedArray::Delete(string object)
{
    long long unsigned int temp, existance_times;
    if (!Find(object, temp, existance_times))
        return false;

    for (long long unsigned int position = temp; position < First_Free_Element - 1; position++)
    {
        Array[position].value = Array[position + 1].value;
        Array[position].times = Array[position + 1].times;
    }

    First_Free_Element--;
    return true;
}
```

Πρόγραμμα 2.4: Δημιουργία της συνάρτησής Delete του αταξινόμητου πίνακα

3 Ταξινομημένος πίνακας

Περιγραφή

Μια δομή ενός ταξινομημένου πίνακα χαρακτηρίζεται από την διάταξη των στοιχείων που ταξινομούνται με κάποια σειρά, όπως αριθμητικό, αλφαβητικό κ.λπ. σε αύξουσα ή φθίνουσα διάταξη. Υπάρχουν πολλοί αλγόριθμοι για την ταξινόμηση ενός λεξικογραφικού πίνακα. Η δομή μας είναι υλοποιημένη με βάση τον αλγόριθμο Insertion Sort.

```
#pragma once
#include <string>

using std::string;

class orderedArray
{
private:
    struct Arr
    {
        string value;
        long long unsigned int times;
    };
    Arr* Array;
    long long unsigned int FirstFreeCell;
protected:
    bool binarySearch(string, long long  int,long long  int, long long int&, long long int&);
    bool ReallocateArray(int);
    void InsertionSort();
public:
    orderedArray();
    ~orderedArray();

    void Insert(string);
    bool Find(string, long long&);
    bool Delete(string);

    long long unsigned int getFirstFreeCell() { return FirstFreeCell; }

    string& operator[](unsigned int possition) { return Array[possition].value; }
};
```

Πρόγραμμα 3.1: Υλοποίηση της κλάσης για τον ταξινομημένο πίνακα

Ξεκινώντας την υλοποίηση της κλάσης, θα πρέπει να διαπραγματευτούμε τις λειτουργίες της μέσα από το header αρχείο της (βλ. Πρόγραμμα 3.1). Η κλάση του ταξινομημένου πίνακα φαίνεται να έχει ελάχιστες διαφορές ως προς τις μεταβλητές και τις συναρτήσεις. Παρακάτω θα αναλυθούν μόνο οι κατά περίπτωση διαφορετικές συναρτήσεις. Οι κοινές υλοποιήσεις – όπως είναι η μέθοδος διαγραφής – φαίνονται στην προηγούμενη δομή.

3.1 Μέθοδος εισαγωγής

Όπως φαίνεται στο παρακάτω τμήμα κώδικα (Πρόγραμμα 3.2) η συνάρτηση Insert δεν διαφέρει πολύ ως προς την υλοποίηση σε σχέση με την Insert του αταξινόμητου πίνακα.

Αρχικά, ελέγχουμε μέσω της Binary Search αν υπάρχει η προς εισαγωγή λέξη ήδη μέσα στον ταξινομημένο πίνακα. Σε περίπτωση που η λέξη δεν υπάρχει καθόλου στον πίνακα, τότε εκτελείται το τμήμα κώδικα που βρίσκεται εντός της else. Το υπόλοιπο τμήμα του κώδικα για τη συνάρτηση Insert είναι κοινό με του αταξινόμητου πίνακα, με τη διαφορά πως μετά την εκχώρηση των τιμών στις μεταβλητές, εκτελείται η συνάρτηση ταξινόμησης του πίνακα (η Insertion Sort που θα αναλύσουμε παρακάτω).

```
void orderedArray::Insert(string word)
{
    long long  int temp ,existance_times;
    if (binary_search(word, 0, LastElement - 1, temp, existance_times))
        Array[temp].times++;
    else
    {
        ReallocateArray(1);
        Array[LastElement - 1].value = word;
        Array[LastElement - 1].times = 1;
        insertionSort();
    }
}
```

Πρόγραμμα 3.2: Δημιουργία της συνάρτησής Insert του ταξινομημένου πίνακα

3.2 Μέθοδος αναζήτησής

Με τη βοήθεια του ταξινομημένου πίνακα, για την αναζήτηση των στοιχείων μπορούμε πλέον να χρησιμοποιήσουμε έναν γρηγορότερο αλγόριθμο για αναζήτηση λέξεων σε σχέση με τον αλγόριθμο σειριακής αναζήτησης.

Η διαδικασία η οποία ακολουθεί ο αλγόριθμος δυαδικής αναζήτησης (Binary Search algorithm) σε έναν ταξινομημένο πίνακα είναι:

1. Ξεκινάμε με ένα διάστημα που καλύπτει ολόκληρο τον πίνακα.
2. Διαιρούμε επανειλημμένα το διάστημα αναζήτησης στο μισό.
3. Εάν η τιμή του κλειδιού αναζήτησης είναι μικρότερη από το στοιχείο στο μέσο του διαστήματος, περιορίζουμε το διάστημα στο κάτω μισό. Διαφορετικά, το περιορίζουμε στο πάνω μισό.
4. Ελέγχουμε επανειλημμένα έως ότου βρεθεί η τιμή ή το διάστημα είναι κενό.

(Ο αλγόριθμος της δυαδικής αναζήτησης φαίνεται στο Πρόγραμμα 3.3)

```
bool orderedArray::binary_search(string word, int left, right, int& temp, int& existance_times)
{
    if (right >= left)
    {
        long long  int mid = left + (right - left) / 2;

        if (Array[mid].value == word)
        {
            temp = mid;
            existance_times = Array[mid].times;
            return true;
        }

        if (Array[mid].value > word)
            return binary_search(word, left, mid - 1, temp, existance_times);

        return binary_search(word, mid + 1, right, temp , existance_times);
    }

    existance_times = 0;
    return false;
}
```

Πρόγραμμα 3.3: Δημιουργία της συνάρτησής Binary Search του ταξινομημένου πίνακα

3.3 Μέθοδος ταξινόμησής

Ο αλγόριθμος Insertion Sort είναι ένας απλός αλγόριθμος ταξινόμησης που λειτουργεί παρόμοιος με τον τρόπο ταξινόμησης των παιγνιοχάρτων. Ο πίνακας χωρίζεται ουσιαστικά σε ένα ταξινομημένο και ένα μη ταξινομημένο τμήμα. Οι τιμές από το μη ταξινομημένο τμήμα επιλέγονται και τοποθετούνται στη σωστή θέση στο ταξινομημένο μέρος.

Σε κάθε επανάληψη, η Insertion Sort αφαιρεί ένα στοιχείο από τα δεδομένα εισαγωγής, βρίσκει τη θέση στην οποία ανήκει στη ταξινομημένη λίστα και το εισάγει εκεί. Η διαδικασία αυτή επαναλαμβάνεται έως ότου δεν υπάρχουν άλλα στοιχεία εισόδου (Βλ. αλγόριθμο σε C++ στο Πρόγραμμα 3.4).

Περίπτωση	Συγκρίσεις	Αντιμεταθέσεις
Καλύτερή	$O(n)$	$O(1)$
Χειρότερή	$O(n^2)$	$O(n^2)$

Πίνακας 3.1: Πολυπλοκότητα του Selection Sort

Για κάθε θέση του πίνακα, ελέγχει την τιμή έναντι της μεγαλύτερης τιμής στη λίστα ταξινόμησης (η οποία συμβαίνει να είναι δίπλα της, δηλαδή στην προηγούμενη θέση του πίνακα που έχει ελεγχθεί). Εάν είναι μεγαλύτερο, αφήνει το στοιχείο στη θέση του και μετακινείται στο επόμενο. Εάν είναι μικρότερο, βρίσκει τη σωστή θέση μέσα στη λίστα ταξινόμησης, μετατοπίζει όλες τις μεγαλύτερες τιμές για να δημιουργήσει κενό και το εισάγει στη σωστή/κενή θέση.

```
void orderedArray::insertionSort()
{
    int i,j,Time;
    string key;
    for(i = 1; i < LastElement; i++)
    {
        key = Array[i].value;
        Time = Array[i].times;
        j = i - 1;
        while(j >= 0 && Array[j].value > key)
        {
            Array[j + 1].value = Array[j].value;
            Array[j + 1].times = Array[j].times;
            j = j - 1;
        }
        Array[j + 1].value = key;
        Array[j + 1].times = Time;
    }
}
```

Πρόγραμμα 3.4: Δημιουργία της συνάρτησής Insertion Sort του ταξινομημένου πίνακα

Ο πίνακας που προκύπτει μετά από i επαναλήψεις έχει την ιδιότητα όπου ταξινομούνται οι πρώτες καταχωρήσεις $i+1$ ($+1$ επειδή η πρώτη καταχώρηση παραλείπεται). Σε κάθε επανάληψη η πρώτη εναπομένουσα καταχώριση της εισόδου αφαιρείται και εισάγεται στο αποτέλεσμα, στη σωστή θέση.

3.4 Σχολιασμός και χρόνοι εκτέλεσης

Ο χρόνος εισαγωγής σε έναν ταξινομημένο πίνακα, είναι αρκετά περισσότερος σε σχέση με την δομή του αταξινόμητου πίνακα που απλά τα στοιχεία εισάγονται στο τέλος. Λόγω της Binary Search που εκτελείται για να ελέγξει αν υπάρχει ήδη το στοιχείο μέσα στον πίνακα και της Insertion Sort που όπως είδαμε στον Πίνακα 3.1 έχει χειρότερη περίπτωση $O(n^2)$, όπου μπορούμε να συμπεράνουμε πως αυτό καθιστά αργή την δομή μας ως προς την εισαγωγή.

Δομή	Χρόνος Εισαγωγής	Χρόνος Αναζήτησης/10χιλ. λέξεις
Ταξινομημένος πίνακας	≈14.3950753 ώρες	≈1.58043333 λεπτά

Πίνακας 3.2: Χρόνοι εκτέλεσης εισαγωγής και αναζήτησης για τον ταξινομημένο πίνακα

Όσο για την αναζήτηση, όπου η διαδικασία που ακολουθείται συμβαίνει αναδρομικά, γνωρίζουμε ότι ο αλγόριθμος της δυαδικής αναζήτησης έχει πολυπλοκότητα $O(log(n))$ που μπορούμε να το διαπιστώσουμε αν αναλύσουμε τον κώδικα του.

Αν εξετάσουμε συνολικά την δομή του ταξινομημένου πίνακα για δέκα χιλιάδες λέξεις, έχουμε τα παραπάνω αποτελέσματα (Πίνακας 3.2).

4 Απλό και δυαδικό δένδρο αναζήτησής τύπου AVL

Περιγραφή

Ένα απλό δυαδικό δέντρο όπως και ένα δένδρο τύπου AVL είναι μία δενδρική δομή δεδομένων η οποία αποτελείται από κόμβους που, όπως και στα δυαδικά δέντρα, έχουν το πολύ δύο παιδιά. Αυτό που καθιστά το δυαδικό δέντρο αναζήτησης διαφορετικό από τα απλά δυαδικά δέντρα είναι το γεγονός ότι οι κόμβοι έχουν πάντα ως αριστερό παιδί αυτό με μικρότερη τιμή σε σχέση με τη ρίζα και δεξί παιδί αυτό με μεγαλύτερη τιμή σε σχέση με τη ρίζα. Το παραπάνω ισχύει – με αναδρομική διαδικασία – για όλα τα υποδέντρα που περιέχονται στο βασικό δυαδικό δέντρο αναζήτησης. Η διαφορά ανάμεσα σε απλό και δένδρο τύπου AVL είναι στα ύψη των δύο θυγατρικών υποδέντρων οποιουδήποτε κόμβου, που διαφέρουν το πολύ κατά ένα.

Όπως προαναφέρθηκε τα δέντρα αποτελούνται από κόμβους, οπότε πριν προχωρήσουμε στις λειτουργίες τις εισαγωγής/αναζήτησής/διαγραφής ας δούμε τι περιέχει το header αρχείο (node.h) που αφορά τους κόμβους που αποθηκεύουμε στο δέντρο (βλ. Πρόγραμμα 4.1).

Αρχικά χρειαζόμαστε ένα struct ώστε να αποθηκεύουμε για κάθε κόμβο τη λέξη που επιθυμούμε και τις φορές εμφάνισης της μέσα στη δομή. Έπειτα ως μέλη της κλάσης χρειαζόμαστε τρεις μεταβλητές τύπου node που αφορούν το δεξί και το αριστερό παιδί κάποιας ρίζας και μια parent που αφορά τον κόμβο-ρίζα που είναι ο γονέας ενός κόμβου για κάποιο υποδέντρο. Τέλος υπάρχουν δύο κατασκευαστές: Ένας κενός που δημιουργεί κενό κόμβο και ένας δεύτερος με όρισμα ένα string που χρησιμοποιείται στην εισαγωγή ενός νέου κόμβου.

Όλα τα παραπάνω είναι δηλωμένα στο δημόσιο τμήμα της κλάσης παρ' όλο που δεν χρησιμοποιούνται στη main . Αυτό συμβαίνει για απλούστευση του κώδικα.

```
#pragma once
#include <string>

using std::string;

struct node_info
{
    string value;
    long long int times;
};

class node
{
public:
    node_info data;
    long long int height;
    node* left;
    node* right;
    node* parent;
    node(const string&);
    node();
};
```

Πρόγραμμα 4.1: Υλοποίηση του struct info και της κλάσης node για τις δενδρικές δομές

4.1 Μέθοδος εισαγωγής

Η εισαγωγή σε ένα απλό δυαδικό δέντρο αναζήτησης και ενός δένδρου τύπου AVL αποτελεί μια λειτουργία/μέθοδο με πολλαπλούς ελέγχους που διευκολύνουν στη συνέχεια τη μέθοδο αναζήτησης και διαγραφής.

Στον κώδικα της μεθόδου εισαγωγής, αρχικά ελέγχεται αν το δέντρο είναι κενό οπότε και δημιουργείται η ρίζα του δέντρου που περιέχει τη λέξη word και η μεταβλητή με τις φορές εμφάνισης τίθεται σε ένα. Για κάθε επόμενο κόμβο που θέλουμε να δημιουργηθεί, ελέγχουμε αν η λέξη (η οποία είναι το κλειδί για τους ελέγχους στην εισαγωγή), είναι μικρότερη ή μεγαλύτερη από τη ρίζα του εκάστοτε δέντρου ή υποδέντρου. Οι έλεγχοι αυτοί επαναλαμβάνονται μέχρι τη στιγμή που θα βρεθεί ένας κόμβος κατάλληλος για την δημιουργία του νέου κόμβου ως παιδί του. Αυτό προϋποθέτει να έχει ο κόμβος-γονέας αριστερό ή δεξί παιδί (ανάλογα τη θέση που πρέπει να εισαχθεί ο νέος κόμβος), ίσο με nullptr . Σε αυτήν την περίπτωση έχουμε επιτυχημένη εισαγωγή και η συνάρτηση insert τερματίζει επιστρέφοντας τη λογική τιμή true . Αξιοσημείωτη είναι η περίπτωση που υπάρχει ήδη ένας κόμβος με αποθηκευμένη λέξη τη word που θέλουμε να

εισάγουμε. Τη περίπτωση αυτή τη διαχειριζόμαστε αυξάνοντας μόνο τον μετρητή που δείχνει τις φορές εμφάνισης της λέξης κατά μια μονάδα. Έτσι βεβαιωνόμαστε ότι κάθε κόμβος του δέντρου αποθηκεύει μόνο τις λέξεις που είναι άγνωστες για τη δομή. Η διαφορά μεταξύ των δυο δένδρων είναι ότι στο τέλος κάθε εισαγωγής, το δένδρο τύπου AVL καλεί τη συνάρτησή εξισορρόπησής αναδρομικά από το κόμβο που εισήχθη έως τη ρίζα του δένδρου, ακολουθώντας το μονοπάτι που ακολουθήθηκε στην insert για την εισαγωγή του.

4.2 Μέθοδος αναζήτησής

Η διαδικασία της αναζήτησης σε ένα απλό δυαδικό δέντρο αναζήτησης όπως και σε ένα δένδρο τύπου AVL δε διαφέρει και πολύ σε σχέση με την δυαδική αναζήτηση σε έναν ταξινομημένο πίνακα.

Ο κόμβος αναζητείται με βάση την λέξη-κλειδί που έχει αποθηκευμένη. Έτσι ξεκινώντας από τη ρίζα του δέντρου, μέσα σε μία επανάληψη ελέγχουμε αν η λέξη-κλειδί για την αναζήτηση είναι μεγαλύτερη ή μικρότερη της λέξης που έχει αποθηκευμένη η ρίζα (αρχικά στον πρώτο έλεγχο), ώστε στην επόμενη επανάληψη να λάβουμε ως κόμβο προς έλεγχο το αριστερό (ή το δεξί) παιδί της ρίζας, αγνοώντας το δεξί (ή το αριστερό υποδέντρο) αντίστοιχα. Η παραπάνω διαδικασία επαναλαμβάνεται μέχρι να βρεθεί ο κόμβος που περιέχει τη λέξη word ή να τελειώσουν οι κόμβοι προς έλεγχο, δηλαδή να έχει γίνει έλεγχος σε όλο το δέντρο και να μη βρέθηκε κόμβος που περιέχει τη λέξη word . Στο τέλος της συνάρτησης search (του ιδιωτικού τμήματος), επιστρέφεται ο κόμβος προς έλεγχο που χρησιμοποιήθηκε στην επανάληψη. Η public search ελέγχει αν ο κόμβος που επιστράφηκε από την private είναι nullptr , δηλαδή δεν βρέθηκε η λέξη word οπότε και επιστρέφει αναφορικά μηδέν. Διαφορετικά επιστρέφεται και πάλι αναφορικά η συχνότητα εμφάνισης της λέξης.

4.3 Μέθοδος διαγραφής

Η διαδικασία της διαγραφής ενός κόμβου από το δέντρο αποτελεί μία περίπλοκη διαδικασία η οποία απλουστεύεται λόγω της χρήσης του κόμβου γονέα (parent*) .

Αρχικά αναζητούμε μέσω της συνάρτησής search που περιγράφεται παρακάτω τον κόμβο που έχει αποθηκευμένη τη λέξη word . Όταν βρεθεί γίνονται οι απαραίτητοι έλεγχοι ώστε να διαπιστωθεί αν ο κόμβος αυτός έχει παιδιά (δηλαδή δεν είναι φύλλο), κι αν ναι πόσα ή είναι η ρίζα του δέντρου. Οι παραπάνω έλεγχοι γίνονται ώστε κατά τη διαδικασία της διαγραφής του κόμβου να γνωρίζουμε ποιος είναι ο γονέας και τα παιδιά του έτσι ώστε ο κόμβους που ενδεχομένως θα πάρει τη θέση του να μη παραβιάζει τις αρχές του δυαδικού δέντρου αναζήτησης σχετικά με τη θέση των κόμβων μέσα σε αυτό.

Στη μέθοδο διαγραφής του AVL υπάρχει μία διαφοροποίηση καθώς η εισαγωγή κόμβου δεν υλοποιείται με τη χρήση του κόμβου γονέα (parent*) . Έτσι όταν θέλουμε να διαγράψουμε έναν κόμβο δεν μπορούμε απλώς αναζητώντας τον να τον διαγράψουμε αφού δεν γνωρίζουμε τον γονέα του. Αυτό το δια χειριστήκαμε εκτελώντας αναδρομική διαδικασία μέσα στη συνάρτηση διαγραφής του AVL προσπαθώντας να βρεθεί το στοιχείο, επαναλαμβάνοντας και παίρνοντας ως κόμβο προς έλεγχο το αριστερό ή δεξί παιδί της ρίζας που ελέγχεται. Όταν και αν βρεθεί σταματούν οι αναδρομικές κλήσεις και ελέγχεται πόσα παιδιά έχει ο κόμβος προς διαγραφή, αν είναι η ρίζα του δέντρου, αν είναι το παιδί κάποιας ρίζας κι αν ναι ποιο. Τότε η διαγραφή γίνεται μια εύκολη διαδικασία κι αφού ολοκληρωθεί, καλείται η συνάρτηση εξισορρόπησης ώστε όλοι οι κόμβοι (από αυτόν που διαγράφηκε έως τη ρίζα) να διατηρούν τις αρχές ισοζυγισμού του δέντρου τύπου AVL .

4.4 Σχολιασμός και χρόνοι εκτέλεσης

Η διαμόρφωση του τελικού απλού δυαδικού δέντρου αναζήτησης οφείλεται στη σειρά εισαγωγής στοιχείων. Γι αυτόν το λόγο παρατηρούμε ότι η δημιουργία του απλού δυαδικού δέντρου αναζήτησης σε σχέση με το AVL απαιτεί λιγότερο χρόνο καθώς επίσης λόγω του ότι δεν υλοποιείται η εξισορρόπηση του δέντρου. Η εξισορρόπηση είναι εμφανώς απαραίτητη στο AVL και απαιτεί περαιτέρω χρόνο για τους ελέγχους που αφορούν τα ύψη των υποδέντρων για κάποια ρίζα κι έπειτα την εξισορρόπηση τους μέσω της λειτουργίας των περιστροφών.

Δομή	Χρόνος Εισαγωγής	Χρόνος Αναζήτησης/10χιλ. λέξεις
Απλό δυαδικό δένδρο	≈4.4155 λεπτά	42.901 δευτερόλεπτα
Δυαδικό δένδρο τύπου AVL	16.0635 λεπτά	28.382 δευτερόλεπτα

Πίνακας 4.1: Χρόνοι εκτέλεσης εισαγωγής και αναζήτησης για τις δυο δεντρικές δομές

Η μέθοδος αναζήτησης είναι η ίδια και για τα δύο δέντρα, απλό δυαδικό δέντρο αναζήτησης και AVL . Παρόλα αυτά στο απλό δυαδικό δέντρο αναζήτησης ενδέχεται το ένα υποδέντρο της ρίζας να είναι κατά πολύ μεγαλύτερο του άλλου, αφού τα στοιχεία εισάγονται με τη σειρά που λαμβάνονται από το αρχείο δίχως περιστροφές ,κι έτσι η λειτουργία της αναζήτησης απαιτεί περισσότερο χρόνο σε σχέση με αυτήν του AVL που όλα τα υποδέντρα έχουν ύψη σε απόλυτη τιμή με διαφορά το πολύ ένα($|\text{Αριστερό υποδέντρο} - \text{δεξί υποδέντρο}| \leq 1$). Στο AVL στην γενική περίπτωση κατά τη διάσχιση του δέντρου για την εύρεση ενός κόμβου αγνοείται μεγαλύτερο πλήθος κόμβων, λόγω του ότι είναι ισορροπημένο γλιτώνοντας χρόνο στη λειτουργία της αναζήτησης.

5 Πίνακας κατακερματισμού με ανοικτή διεύθυνση

Περιγραφή

Ενώ ο στόχος μιας συνάρτησης κατακερματισμού είναι η ελαχιστοποίηση των συγκρούσεων, ορισμένες συγκρούσεις είναι αναπόφευκτες στην πράξη. Έτσι, οι εφαρμογές κατακερματισμού πρέπει να περιλαμβάνουν κάποια μορφή πολιτικής επίλυσης σύγκρουσης. Οι τεχνικές επίλυσης σύγκρουσης μπορούν να χωριστούν σε δύο κατηγορίες: ανοιχτό κατακερματισμό και κλειστό κατακερματισμό (ονομάζεται επίσης ανοιχτής διεύθυνσης). Η διαφορά μεταξύ των δύο έχει να κάνει με το εάν οι συγκρούσεις αποθηκεύονται εκτός πίνακα (δηλαδή με συνδεδεμένες λίστες) ή εάν οι συγκρούσεις έχουν ως αποτέλεσμα την αποθήκευση στον ίδιο πίνακα.

Ωστόσο, το θέμα των συγκρούσεων θα μπορούσε να βελτιωθεί, αν και αυτό δεν σημαίνει ότι θα αποτρέψουμε όλες τις συγκρούσεις, αλλά μπορούμε να μειώσουμε την πιθανότητα δυο λέξεις να συγκρουστούν, δηλαδή να έχουν το ίδιο hash key . Μια καλή και ευρέως χρησιμοποιούμενη συνάρτηση υπολογισμού του hash μιας συμβολοσειράς βρίσκεται παρακάτω (με κάποιες βελτιώσεις - βάση του αλγορίθμου μας).

Έστω s μια συμβολοσειρά και q το μέγεθος του πίνακα. Έχουμε:

$$\begin{aligned} hash(s) &= \frac{1}{q}\left(s[0] + s[1]\cdot p + s[2]\cdot p^2 + \ldots + s[n-1]\cdot p^{n-1} \mod m\right) \\ &= \frac{1}{q}\left(\sum_{i=0}^{n-1} s[i]\cdot p^i \mod m\right) \end{aligned}$$

όπου $p = 31$ και $m = 1e9 + 9$ [1].

```cpp
#pragma once
#include <string>

using std::string;

class HashNode
{
public:
    HashNode(string word, int key, int times)
    {
        this->word = word;
        this->key = key;
        this->times = times;
    }
    string word;
    unsigned int key, times;
};
```

Πρόγραμμα 5.1: Υλοποίηση του HashNode για τον πίνακα κατακερματισμού με ανοιχτή διεύθυνση

Για την υλοποίηση της δομής η λογική που ακολουθήθηκε είναι να δημιουργήσουμε ένα Hash Node όπου θα αποθηκεύουμε την λέξη, το κλειδί και τις φορές που υπάρχει μέσα στην δομή. Ουσιαστικά, με τη βοήθεια του Hash Node θα μπορούμε να διαχειριζόμαστε δυναμικά, γρηγορότερα και ευκολότερα την δομή μας.

Προχωρώντας στην υλοποίηση της κεντρικής κλάσης για τον πίνακα κατακερματισμού με ανοιχτή διεύθυνση, αρχικά, ορίζουμε ότι θέλουμε να χρησιμοποιήσουμε μια μεταβλητή που θα αποθηκεύει τον χώρο, μια τα στοιχεία τα οποία εισήχθησαν και τέλος έναν πίνακα τύπου HashNode (βλ. Πρόγραμμα 5.1).

Στο protected τμήμα της κλάσης έχουμε την συνάρτηση getKey , όπου μετατρέπει μια λέξη σε αριθμό, την Get όπου επιστρέφει συγκεκριμένη διεύθυνση του πίνακα, την Hash όπου υπολογίζει το κλειδί και τέλος την nextHash όπου υπολογίζουμε το επόμενο κλειδί.

Στο public τμήμα της κλάσης έχουμε τον κατασκευαστή, την συνάρτηση όπου θα εισάγει λέξεις μέσα στη δομή και την συνάρτηση που θα αναζητεί (βλ. Πρόγραμμα 5.2).

[1] Περισσότερα για την συνάρτησή στο άρθρο: https://cp-algorithms.com/string/string-hashing.html

```
#pragma once
#include <string>

using std::string;

class Hashing
{
private:
    unsigned long long int Size;
    unsigned long long int Elements;
    HashNode** Table;
protected:
    long long getKey(const string&);
    HashNode*& Get(const int&, const string&);
    unsigned int Hash(const int&);
    unsigned int nextHash(unsigned int);
public:
    Hashing();
    ~Hashing();
    bool Insert(const string&);
    bool Find(string&, unsigned int &);
};
```

Πρόγραμμα 5.2: Υλοποίηση της κλάσης για τον πίνακα κατακερματισμού με ανοιχτή διεύθυνση

5.1 Μέθοδος εισαγωγής

Η σκεπτική πλευρά της μεθόδου εισαγωγής προγραμματιστικά θα λέγαμε ότι είναι απλή (βλ. Πρόγραμμα 5.3). Αρχικά, ελέγχουμε αν η διεύθυνση η οποία μας επέστρεψε η συνάρτηση Get (ουσιαστικά, η διεύθυνση μιας συγκεκριμένης θέσης του πίνακα - με βάση το key και την λέξη) είναι κενή ή όχι. Σε περίπτωση που χρησιμοποιείται τότε επιστρέφουμε false , δηλαδή ότι δεν μπόρεσε η συγκεκριμένη λέξη να εισαχθεί. Διαφορετικά προχωρούμε κανονικά ελέγχοντας αν μας επαρκεί η μνήμη που έχουμε. Αν έχει ήδη γεμίσει το μισό του πίνακα τότε διπλασιάζουμε τον χώρο με κύριο κριτήριο οι λέξεις να εισαχθούν στον νέο πίνακα με την περισσότερη μνήμη με βάση την συνάρτησή Hash , για να μπορούμε οποιαδήποτε στιγμή να αναζητήσουμε οποιοδήποτε στοιχείο επιθυμούμε. Στη συνέχεια εκχωρούμε την λέξη. Η περίπτωση μια λέξη να υπάρχει ήδη στον πίνακα ελέγχεται από την συνάρτηση Get την οποία θα δούμε στο κεφάλαιο με την μέθοδο αναζήτησης.

5.2 Μέθοδος αναζήτησής

Όπως φαίνεται και στο Πρόγραμμα 5.4, η συνάρτηση Find δέχεται δύο ορίσματα. Το πρώτο όρισμα δέχεται τη λέξη την οποία θέλουμε να εξετάσουμε αν υπάρχει μέσα στη δομή (η συνάρτηση δέχεται τη λέξη αναφορικά για να μη γίνει άσκοπη αντιγραφή) και το δεύτερο όρισμα τη μεταβλητή που χρησιμοποιούμε για να επιστρέψουμε το σύνολο εμφανίσεων της κάθε λέξης στο σύνολο όλων των λέξεων που έχει αποθηκεύσει η δομή. Η επιστροφή του συνόλου εμφανίσεων μιας λέξης γίνεται με την χρήση αναφορικής μεταβλητής.

Εντός της συνάρτησης, αρχικά ορίζουμε σε μια μεταβλητή το HashKey που επιστρέφεται από την συνάρτηση Hash και έπειτα δημιουργούμε μια μεταβλητή που θα αποθηκεύει διεύθυνση που επιστρέφει η συνάρτηση Get . Ο τρόπος λειτουργίας της συνάρτησης Get είναι ο εξής:

1. Σε μια μεταβλητή p, αποθήκευσε το Hash Key .
2. Καθώς κάθε θέση του πίνακα έχει στοιχεία εξέτασε για κάθε θέση του πίνακα αν η λέξη που είναι αποθηκευμένη είναι ίδια με αυτή που υπάρχει ως παράμετρο. Αν ισχύει, τότε αύξησε το πλήθος της λέξης κατά μια μονάδα και επέστρεψε τη θέση του πίνακα με τις ιδιότητες του. Διαφορετικά, πήγαινε στην επόμενη θέση με βάση τη συνάρτηση nextHash .
3. Σε περίπτωση που δεν ισχύει το παραπάνω, τότε επιστρέφουμε τη θέση του πίνακα με τις ιδιότητες της.

Στη συνέχεια, αφού έχει κληθεί και έχει επιστραφεί κάποιο αποτέλεσμα από την συνάρτηση Get , εξετάζουμε αν η μεταβλητή p δείνχει σε κάποια διεύθυνση θέσης του πίνακα ή όχι. Αν δεν δείχνει κάπου τότε επιστρέφουμε false , διαφορετικά με αναφορά εκχωρούμε τις τιμές στις αντίστοιχες αναφορικές μεταβλητές times και word . Τέλος επιστρέφουμε true .

5.3 Σχολιασμός και χρόνοι εκτέλεσης

Ο χρόνος εισαγωγής σε έναν έναν πίνακα κατακερματισμού είναι λιγότερος σε σχέση με όλες τις προηγούμενες δομές. Ο καθορισμός της θέσης που θα αποθηκευτεί το κάθε στοιχείο ορίζεται πολυωνυμικά με βάση την παραπάνω συνάρτηση που υπάρχει στην περιγραφή. Σε περίπτωση, όπως αναφέρθηκε, που δεν βρεθεί ψάχνει σε άλλη – επόμενη – θέση, με βάση τη συνάρτηση nextHash (βλ. Πρόγραμμα 5.4) όπου δεν ΄κοστίζει΄ πολύ σε πολυπλοκότητα.

Δομή	Χρόνος Εισαγωγής	Χρόνος Αναζήτησης/10χιλ. λέξεις
Πίνακας κατακερματισμού με ανοιχτή διεύθυνση	48.971 δευτερόλεπτα	7.552 δευτερόλεπτα

Πίνακας 5.1: Χρόνοι εκτέλεσης εισαγωγής και αναζήτησης για τον πίνακα κατακερματισμού με ανοιχτή διεύθυνση

Όσο για την αναζήτηση, όπου η διαδικασία που ακολουθείται που ακολουθείτε είναι ακριβώς η ίδια (Αλγοριθμικά περιγράφεται στο Πρόγραμμα 5.4) ο χρόνος σε σχέση με τις προηγούμενες τέσσερις δομές έχει μεγάλη απόκλιση, λόγω της δομής του πίνακα κατακερματισμού. Μεγάλη βελτίωση στην εισαγωγή πραγματοποίησε η συνάρτηση όπου αναφέρεται στην περιγραφή, λόγω λιγότερων συγκρούσεων.

```
bool Hashing::Insert(const string& word)
{
    int key = getKey(word);

    if (Get(key, word) != nullptr)
        return false;

    if (Elements * 2 >= Size)
    {
        Size *= 2;
        HashNode** temp;

        temp = new (std::nothrow) HashNode*[Size];
        for (int i = 0; i < Size; i++)
            temp[i] = nullptr;

        for (int i = 0; i < Size / 2; i++)
        {
            if (Table[i] != nullptr)
            {
                unsigned int p = Hash(Table[i]->key);

                while (temp[p] != nullptr)
                    p = nextHash(p);

                temp[p] = Table[i];
                Table[i] = nullptr;
            }
        }

        delete[] Table;
        Table = temp;
    }

    unsigned int HashIndex = Hash(key);
    while (Table[HashIndex] != nullptr)
        HashIndex = nextHash(HashIndex);

    Table[HashIndex] = new HashNode(word, key, 0); Elements++;
    return true;
}
```

Πρόγραμμα 5.3: Δημιουργία της συνάρτησής Insert του πίνακα κατακερματισμού με ανοιχτή διεύθυνση

```
bool Hashing::Find(string& word, unsigned int& times)
{
    int key = getKey(word);
    HashNode* p = Get(key, word);

    if (p == nullptr)
        return false;

    times = p->times;
    word = p->word;

    return true;
}

HashNode*& Hashing::Get(const int& key, const string& word)
{
    unsigned int p = Hash(key);
    while (Table[p] != nullptr)
    {
        if (Table[p]->word == word
        {
            Table[p]->times++;
            return Table[p];
        }

        p = nextHash(p);
    }

    return Table[p];
}
```

Πρόγραμμα 5.4: Δημιουργία της συνάρτησής Find και Get του πίνακα κατακερματισμού με ανοιχτή διεύθυνση

Τεχνικές λεπτομέρειες

Στον Πίνακα 5.2 θα βρείτε μερικές χρήσιμες πληροφορίες σχετικά με το πρότζεκτ. Όπως αναφέρεται στον παρακάτω πίνακα, το πρότζεκτ έχει δοκιμαστεί επιτυχώς στις παρακάτω τρεις πλατφόρμες-προγράμματα (IDE και online IDE).

Γλώσσα	Έκδοση	Δοκιμάστηκε
C++	C++14	Repl.it (Μικρό αρχείο), Visual Studio 2019 και *Clion 2021.1.1 (Μεγάλο αρχείο)

Πίνακας 5.2: Πίνακας με βασικές τεχνικές λεπτομέρειες

*Οι δομές δοκιμάστηκαν με το πρόγραμμα Clion 2021.1.1 (για το μεγάλο αρχείο) σε λειτουργικό σύστημα Linux (Ubuntu 20.04 LTS) και Windows 10 Home (Έκδοση 20H2) .