Εργασία εξαμήνου στις Δομές Δεδομένων

Η εργασία του 2ου εξαμήνου για το μάθημα Δομές Δεδομένων μας ζητούσε να υλοποιήσουμε 5 δομές.

1. *Αταξινόμητο πίνακα*
2. *Ταξινομημένο πίνακα*
3. *Δυαδικό δέντρο αναζήτησης*
4. *Δυαδικό δέντρο αναζήτησης AVL*
5. *Πίνακας Κατακερματισμού*

Στόχος ήταν να καταφέρουμε να υλοποιήσουμε τις 5 δομές δίχως την χρήση εξωτερικών βιβλιοθηκών που τις περιέχουν έτσι ώστε να καταλάβουμε καλύτερα το πως λειτουργούν και πότε να επιλέγουμε τη κάθε δομή ανάλογα με το τι θέλουμε να κάνουμε.

Κάθε δομή διαφέρει στους χρόνους εισαγωγής και αναζήτησης οπότε μπορούμε σχετικά εύκολα να αναγνωρίσουμε την ταχύτερη δομή καθώς και την πιο αργή. Η πιο αργή είναι ο αταξινόμητος πίνακας, ο οποίος μπορεί να θεωρηθεί και η πιο εύκολη δομή από άποψη δυσκολίας της υλοποίησης της, ενώ η πιο γρήγορη είναι ο πίνακας κατακερματισμού, τόσο από άποψη ταχύτητας εισαγωγής, όσο και αναζήτησης.

Κομμάτι της εργασίας ήταν επίσης η εισαγωγή στο πρόγραμμα των λέξεων ενός αρχείου txt καθώς και η μέτρηση των εμφανίσεων των λέξεων στο σε αυτό το αρχείο καθώς και η χρήση ενός τυχαίου συνόλου λέξεων για την μέτρηση του χρόνου αναζήτησης.

A. Main

Στην main, γίνεται το διάβασμα των λέξεων από το txt , η επεξεργασία αυτών, η εισαγωγή τους στις δομές ,η μέτρηση του χρόνου διεξαγωγής της αναζήτησης (Search) σε κάθε δομή καθώς και η εγγραφή των αποτελεσμάτων στο αρχείο output.txt.

Οι βιβλιοθήκες και τα αρχεία που χρησιμοποιούνται είναι τα παρακάτω:

```
#include <iostream>
#include <fstream>
#include <chrono>
#include "UnsortedArray.h"
#include "SortedArray.h"
#include "BinarySearchTree.h"
#include "HashTable.h"
#include "AVLTree.h"

using namespace std;
using namespace std::chrono;
```

Η fstream χρησιμοποιείται για την διαχείριση αρχείων και η chrono για την μέτρηση χρόνων.

Αρχικά γίνονται οι απαραίτητες δηλώσεις και αρχικοποιήσεις μεταβλητών ,και έπειτα ανοίγουμε το αρχείο που θέλουμε μια φορά ώστε να μετρήσουμε τις λέξεις που έχει ,έτσι ώστε να ξέρουμε τι μέγεθος θα δώσουμε στον πίνακα string που θα έχει αποθηκευμένες τις λέξεις μέσα στην main.

Αυτό γίνεται μέσω της συνάρτησης countWords, η οποία δέχεται ως όρισμα μια μεταβλητή ifstream (αρχείο προς ανάγνωση) και όσο μπορεί να εισάγει λέξεις σε μια string μεταβλητή αυξάνει το count των λέξεων, το οποίο στο τέλος επιστρέφει.

Η επεξεργασία των λέξεων γίνεται κατά την ανάγνωσή τους από το αρχείο. Για να γίνει αυτή χρησιμοποιείται η deleteSpecialChars η οποία αφού διαγράψει όλους τους ειδικούς χαρακτήρες που περιέχονται στην λέξη, επιστρέφει την νέα επεξεργασμένη λέξη, η οποία εισάγεται στον πίνακα string της main. Η deleteSpecialChars είναι bool συνάρτηση η οποία αν μετά την μετά την αφαίρεση των ειδικών χαρακτήρων από μία λέξη, η λέξη μείνει κενή, τότε γυρνάει false ώστε να μειωθεί ο αριθμός των λέξεων που ξέρουμε ότι έχει το αρχείο, γεγονός που μας βοηθά στην Insert ώστε να μην εισάγουμε κενές συμβολοσειρές στις δομές.

Για να διαγράφει τους ειδικούς χαρακτήρες η deleteSpecialChars διαγράφει τον πρώτο ειδικό χαρακτήρα που βρίσκει και πηγαίνει μετά τον μετρητή 1 θέση πίσω έτσι ώστε αν ένας νέος ειδικός χαρακτήρας μεταφέρθηκε στη θέση του παλιού, να διαγράψει και αυτόν.

Έπειτα μέσω μίας for loop εισάγονται οι λέξεις σε όλες τις δομές μέσω της εκάστοτε συνάρτησης Insert για κάθε δομή και μετράμε πόσο χρόνο παίρνει αυτή η διαδικασία. Δημιουργούμε ταυτόχρονα έναν νέο πίνακα q με στόχο να βάλουμε τυχαίες λέξεις για να ελέγξουμε τις αναζητήσεις. Το γεγονός ότι το σύνολο είναι τυχαίο εξασφαλίζεται από την srand μαζί με τη χρήση της time, άρα για κάθε διαφορετική χρονική στιγμή έχουμε διαφορετική αλληλουχία λέξεων στην rand() που χρησιμοποιείται μετά. Επίσης ανοίγουμε το “output.txt” για εγγραφή.

Τέλος, για να βρούμε τον χρόνο του Search χρησιμοποιούμε τον παρακάτω κώδικα για κάθε δομή βάζοντας κάθε φορά το όνομα κάθε δομής (UA,SA,BST,AVL,HT). Το screenshot είναι παράδειγμα του κώδικα για τον Unsorted Array.

```
// Search in the unsorted array
start = high_resolution_clock::now();
for (i = 0; i < searchsize; i++)
{
    if (UA.Search(q[i])!=-1)
        ofs << "Found the word \"" << q[i] << "\" " << UA.getCount(q[i]) << " times in the unsorted array." << endl;
}
stop = high_resolution_clock::now();
duration = duration_cast<milliseconds>(stop - start);
ofs << "The search of the unsorted array took " << duration.count() << " milliseconds." << endl;
```

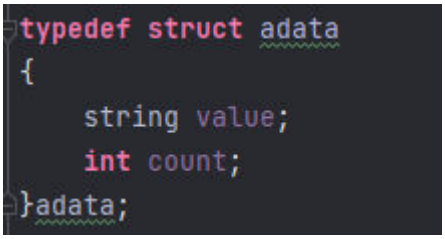
Όπως θα αναφερθεί και παρακάτω η ανεπιτυχής αναζήτηση σε κάθε δομή επιστρέφει την τιμή -1 οπότε αν το αποτέλεσμα της αναζήτησης που κάνουμε είναι διάφορο από αυτό τότε γράφουμε στο αρχείο ότι βρήκαμε την λέξη και εμφανίζουμε και το πόσες φορές εμφανίστηκε με την εκάστοτε getCount(string) κάθε δομής.

Β. Αταξινόμητος πίνακας (Unsorted Array)

Ο αταξινόμητος πίνακας είναι αδιαμφισβήτητα η πιο αργή δομή. Το γεγονός αυτό οφείλεται στη φύση του. Από τη στιγμή που δεν υπάρχει ταξινόμηση και κάθε στοιχείο μπορεί να βρίσκεται σε οποιοδήποτε κελί του πίνακα, η μόνη αναζήτηση που μπορούμε

να χρησιμοποιήσουμε είναι η σειριακή, η οποία είναι και η πιο αργή αναζήτηση με πολυπλοκότητα χειρότερης περίπτωσης (worst case) O(n), μπορεί δηλαδή να αναγκαστεί το πρόγραμμα να ψάξει όλες τις λέξεις του πίνακα.

Ένα struct που χρησιμοποιούμε και σε ορισμένες άλλες δομές είναι το παρακάτω struct adata έτσι ώστε η τιμή ενός κελιού και ο αριθμός των εμφανίσεων της τιμής να είναι ένα ζευγάρι.



Στις protected μεταβλητές (θα κληρονομεί από την κλάση UnsortedArray η κλάση SortedArray) υπάρχει ένας adata pointer aData και μια int μεταβλητή size.

Public είναι οι 2 κατασκευαστές (κενός που αρχικοποιεί το size σε 0 και με int όρισμα που αρχικοποιεί το size με την int τιμή), οι συναρτήσεις Insert, Search, Delete και Print καθώς και οι συναρτήσεις getCount που γυρνάει τον αριθμό εμφανίσεων μιας λέξης και η getWords που εμφανίζει το σύνολο των μοναδικών λέξεων στον πίνακα.

Η Insert ελέγχει πρώτα αν το size είναι 0 ώστε να κάνει την πρώτη εισαγωγή. Για τις άλλες περιπτώσεις ορίζει μια bool μεταβλητή exists, η οποία αν η λέξη υπάρχει στον πίνακα παίρνει την τιμή true αλλιώς την τιμή false. Αν η exists είναι true αυξάνουμε την μεταβλητή count της λέξης κατά 1 ,αλλιώς, σε έναν προσωρινό πίνακα που δημιουργούμε κάθε 1000 λέξεις -ή μετά τη πρώτη εισαγωγή-, προσθέτουμε τη νέα λέξη και ορίζουμε τον count της σε 1 εξασφαλίζοντας έτσι ότι όσες λέξεις εισάγονται έχουν count ίσο με 1 και όχι κάποια τυχαία τιμή. Η αντιγραφή των δεδομένων του aData στον temp πίνακα γίνεται με for loop και το size αυξάνεται κατά 1 κάθε φορά που προσθέτουμε μία λέξη.

Η Search είναι μια απλή σειριακή αναζήτηση μιας string η οποία αν βρει τη string που ψάχνουμε επιστρέφει τη θέση της, αλλιώς γυρνάει -1.

Η Delete ψάχνει την λέξη που θέλουμε να αντιγράψουμε και αν τη βρει, μετακινεί όλα τα στοιχεία μια θέση αριστερά διαγράφοντας αυτό που θέλουμε και δημιουργεί έναν νέο πίνακα adata με μια λιγότερη θέση στον οποίο αντιγράφει τον πίνακα aData. Ταυτόχρονα μειώνει τη τιμή του size κατά 1.

Η Print απλά εκτυπώνει όλες τις λέξεις μαζί με το πόσες φορές εμφανίζονται.

Η getCount ψάχνει τη λέξη μέσω της Search και γυρνάει τον count εκείνης της θέσης (αφού η Search γυρνάει θέση), ενώ η getWords επιστρέφει το size του πίνακα.

C. Ταξινομημένος πίνακας (Sorted Array)

Ο ταξινομημένος πίνακας είναι μια αρκετά γρήγορη δομή για αναζητήσεις (πολύ πιο γρήγορη από τον αταξινόμητο) . Αυτό ισχύει διότι για τις αναζητήσεις που χρειάζονται τόσο για την Search όσο και για την Insert και τη Delete μπορεί να χρησιμοποιηθεί ο αλγόριθμος της δυαδικής αναζήτησης (binary search) ο οποίος έχει πολυπλοκότητα χειρότερης περίπτωσης (worst case) O(logn), είναι δηλαδή πολύ πιο γρήγορος από την σειριακή αναζήτηση.

Το γεγονός ότι μπορούμε να χρησιμοποιήσουμε δυαδική αναζήτηση οφείλεται στο ότι ο πίνακας είναι ταξινομημένος και παραμένει ταξινομημένος μετά από την εισαγωγή κάθε λέξης. Αυτό επιτυγχάνεται μέσω της δυαδικής αναζήτησης.

```
int binarysearch(adata* a,int l,int h,string x)
{
    int mid;
    if (h>=l)
    {
        mid = l + (h - l) / 2;
        if (a[mid].value == x)
            return mid;
        if (a[mid].value > x)
            return binarysearch(a, l, mid - 1, x);
        if (a[mid].value < x)
            return binarysearch(a, mid+1, h, x);
    }
    return l;
}
```

Η δυαδική αναζήτηση βρίσκει κάθε φορά ένα μεσαίο στοιχείο και ελέγχει αν το στοιχείο που ψάχνουμε είναι μεγαλύτερο ή μικρότερο από αυτό. Αν είναι μεγαλύτερο επαναλαμβάνει την ίδια διαδικασία στο δεξιό κομμάτι του πίνακα ενώ αν είναι μικρότερο, στο αριστερό. Αυτή η διαδικασία επαναλαμβάνεται αναδρομικά είτε μέχρι να βρεθεί το στοιχείο, είτε μέχρι μια αναδρομική κλήση της συνάρτησης να κληθεί με την μεταβλητή h (high) να είναι μικρότερη της l (low). Σε κείνη την περίπτωση επιστρέφει την μεταβλητή low η οποία είναι και η θέση στην οποία θα έπρεπε να βρίσκεται το στοιχείο που ψάχνουμε αν υπήρχε στον πίνακα.

Η κλάση SortedArray κληρονομεί από την UnsortedArray και έτσι οι private μεταβλητές της είναι ίδιες με τις protected της Unsorted, το ίδιο και οι κατασκευαστές, και οι συναρτήσεις Print(),getWords().

Η getCount() είναι ίδια απλά χρησιμοποιεί την Search του Sorted ώστε να χρησιμοποιεί δυαδική αναζήτηση.

Στην Insert, χρησιμοποιείται και πάλι η bool μεταβλητή exists και αν είναι true (μετά από δυαδική αναζήτηση της λέξης), αυξάνουμε τον count της λέξης κατά 1. Αλλιώς ακολουθούμε αντίστοιχη διαδικασία με αυτή της Insert του αταξινόμητου πίνακα με τη διαφορά ότι το στοιχείο δεν τοποθετείται στο τέλος του πίνακα, αλλά στην θέση που αντιστοιχεί στο αποτέλεσμα της binarysearch, έτσι ώστε ο πίνακας να παραμένει πάντα ταξινομημένος. Τοποθετείται συνεπώς το στοιχείο στην θέση που πρέπει και όλα τα επόμενα μετακινούνται μία θέση δεξιά.

Η Search κάνει δυαδική αναζήτηση στον πίνακα aData. Αν το στοιχείο που επιστρέψει έχει τιμή ίδια με την string που ψάχνουμε τότε επιστρέφει την θέση του στοιχείου. Στην άλλη περίπτωση επιστρέφει -1.

Η Delete είναι ακριβώς ίδια με την Delete του αταξινόμητου με τη διαφορά ότι για να βρούμε την λέξη (αν υπάρχει) χρησιμοποιούμε την Search του ταξινομημένου (δυαδική αναζήτηση).

D. Δυαδικό δέντρο αναζήτησης (Binary search tree)

Το Δυαδικό Δένδρο Αναζήτησης (ΔΔΑ) αποτελείται, όπως και κάθε δένδρο, από κόμβους και ακμές που τους συνδέουν μεταξύ τους. Κάθε κόμβος έχει έναν γονιό, εκτός από τη ρίζα, και παιδιά, εκτός από τα φύλλα. Στα ΔΔΑ κάθε κόμβος έχει το πολύ δύο παιδιά. Το αριστερό υποδένδρο κάθε κόμβου στο ΔΔΑ έχει τιμές μικρότερες από την τιμή του δένδρου και το δεξί μεγαλύτερες τιμές από την τιμή του κόμβου. Τα ΔΔΑ, όπως φανερώνει και το όνομά τους, χρησιμοποιούνται στην αναζήτηση στοιχείων, καθώς επιταχύνεται η λειτουργία της αναζήτησης σημαντικά. Για το ΔΔΑ του προγράμματος πραγματοποιούνται ορισμένες διεργασίες, μεταξύ άλλων η εισαγωγή, η διαγραφή και η αναζήτηση κόμβων. Αναλυτικότερα, στο Δυαδικό Δένδρο Αναζήτησης περιέχονται:

Ο κενός κατασκευαστής node που αρχικοποιεί τους δείκτες left, right και parent με nullptr και την μεταβλητή count με μηδέν(0).

Ο κατασκευαστής node που δέχεται ένα string και δίνει στις μεταβλητές value το string που δέχτηκε και count το 1, ενώ ξαναορίζει τους δείκτες left, right, parent με nullptr.

Ο κενός κατασκευαστής BinarySearchTree που αρχικοποιεί τον δείκτη root με nullptr.

Κάθε συνάρτηση του BST έχει μια public έκδοση έτσι ώστε αυτή να καλείται στην main

Η λογική συνάρτηση Insert που δέχεται έναν κόμβο r και το string. Η συγκεκριμένη συνάρτηση είναι αναδρομική και κάθε φορά ελέγχει αν η τιμή του κόμβου είναι ίδια με το string. Αν ισχύει τότε αυξάνεται η μεταβλητή count της λέξης κατά ένα και επιστρέφει false.. Στη συνέχεια, εξετάζεται αν η λέξη είναι μεγαλύτερη ή μικρότερη λεξικογραφικά από την τιμή του κόμβου r. Αν είναι μεγαλύτερη, τότε εφόσον ο κόμβος δεν έχει δεξί παιδί, δημιουργείται, λαμβάνει την τιμή string, ορίζεται γονιός του νέου κόμβου ο κόμβος r που είχε δεχθεί η συνάρτηση ως όρισμα και η συνάρτηση επιστρέφει την λογική τιμή true. Αν υπάρχει το παιδί τότε ξανακαλείται αναδρομικά η λογική συνάρτηση με ορίσματα το δεξί παιδί του κόμβου και το string. Η παρόμοια διαδικασία πραγματοποιείται και στην

περίπτωση που η λέξη είναι μικρότερη από την τιμή του κόμβου.

Η συνάρτηση search τύπου node* η οποία δέχεται ως ορίσματα έναν κόμβο p και μία λέξη. Όσο ο κόμβος p δεν είναι nullptr και η τιμή του είναι διάφορη του string τότε επαναλαμβάνεται η διαδικασία αναζήτησης της λέξης με τον κόμβο p να αλλάζει σε κάθε επανάληψη καθώς πλησιάζουμε προς τα φύλλα του δένδρου. Η συνάρτηση επιστρέφει τον κόμβο p. Στην public Search αν ο κόμβος p είναι nullptr επιστρέφεται η τιμή -1 αλλιώς η τιμή 1.

Η συνάρτηση inOrder τύπου void που δέχεται την ρίζα(root) του δένδρου και ελέγχει αν είναι nullptr. Στην περίπτωση που είναι τότε επιστρέφει αλλιώς εμφανίζει το δένδρο σύμφωνα με την ενδοδιατεταγμένη διάσχιση.

Η συνάρτηση preOrder τύπου void που δέχεται την ρίζα(root) του δένδρου και ελέγχει αν είναι nullptr. Στην περίπτωση που είναι τότε επιστρέφει αλλιώς εμφανίζει το δένδρο σύμφωνα με την προδιατεταγμένη διάσχιση.

Η συνάρτηση postOrder τύπου void που δέχεται την ρίζα(root) του δένδρου και ελέγχει αν είναι nullptr. Στην περίπτωση που είναι τότε επιστρέφει αλλιώς εμφανίζει το δένδρο σύμφωνα με την μεταδιατεταγμένη διάσχιση.

Η ακέραια συνάρτηση getHeight η οποία δέχεται έναν κόμβο r και είναι αναδρομική. Ελέγχει κάθε φορά αν ο κόμβος είναι nullptr. Αν είναι, επιστρέφει μηδέν(0). Διαφορετικά, καλείται αναδρομικά έως ότου φτάσει στα φύλλα του δένδρου. Με κάθε επιστροφή της αυξάνει της μεταβλητές στις οποίες κλήθηκε κατά ένα και πριν την τελευταία της επιστροφή συγκρίνει τις μεταβλητές και επιστρέφει την μεγαλύτερη.

Η λογική συνάρτηση Delete με όρισμα τον κόμβο child αρχικά ελέγχει αν ο κόμβος είναι nullptr. Αν είναι επιστρέφει false. Αν όχι, δημιουργεί την μεταβλητή parent και η οποία αναπαριστά τον γονιό του κόμβου child. Στη συνέχεια, διακρίνονται τρεις περιπτώσεις:

1. Ο κόμβος να μην έχει παιδιά. Σε αυτήν την περίπτωση, εξετάζεται αν ο κόμβος είναι η ρίζα. Αν είναι γίνεται διαγραφή της ρίζας και το ρ παίρνει την τιμή nullptr. Αν ο κόμβος είναι το αριστερό ή το δεξί παιδί του γονιού τότε διαγράφεται ο κόμβος και ο αριστερός ή ο δεξιός δείκτης του γονιού αντίστοιχα γίνεται nullptr.
2. Ο κόμβος να έχει ένα παιδί. Αν ο κόμβος είναι η ρίζα τότε τη θέση της την παίρνει το παιδί και ύστερα διαγράφεται ο κόμβος. Αν το ο κόμβος είναι το αριστερό ή το δεξί παιδί του γονιού του τότε ο αριστερός ή δεξιός δείκτης του γονιού δείχνει πλέον στο παιδί του παιδιού του και ο κόμβος child διαγράφεται.
3. Ο κόμβος έχει δύο παιδιά. Στην περίπτωση αυτή αναζητείται ο κόμβος του δεξιού υποδένδρου του child με την μικρότερη τιμή, η οποία αντικαθιστά την τιμή του child και στη συνέχεια διαγράφεται ο κόμβος του δεξιού υποδένδρου του child με τιμή ίδια με την νέα τιμή του child.

Ε. Δυαδικό δέντρο αναζήτησης AVL

Το AVL είναι ένα ισοζυγισμένο δυαδικό δέντρο αναζήτησης, του οποίου η διαφορά των υψών μεταξύ αριστερού και δεξιού υποδένδρου κάθε κόμβου δεν μπορεί να είναι μεγαλύτερη του 1 ή μικρότερη του -1. Το AVL δέντρο του προγράμματος κληρονομεί τις συναρτήσεις της κλάσης Binary Search Tree του προγράμματος. Αναλυτικότερα, στο AVL δένδρο περιέχονται:

Η ακέραια συνάρτηση Balance Factor, η οποία δέχεται ως όρισμα έναν κόμβο r. Αρχικά, ελέγχει αν ο κόμβος r είναι nullptr. Αν είναι επιστρέφει 0 , σε αντίθετη περίπτωση επιστρέφει τη διαφορά των υψών του αριστερού υποδένδρου με το δεξί υποδένδρο.

Η συνάρτηση Left Rotation, η οποία είναι τύπου node* και δέχεται έναν κόμβο r. Στη συνάρτηση αρχικοποιείται ο κόμβος

child με τον δείκτη του δεξιού παιδιού του κόμβου r και ο δείκτης T με τον δείκτη του αριστερού παιδιού του κόμβου child. Στη συνέχεια, το r γίνεται αριστερό παιδί του κόμβου child και το Τα δεξί παιδί του κόμβου r. Η συνάρτηση επιστρέφει τη νέα ρίζα child.

Η συνάρτηση Right Rotation που είναι τύπου node* και δέχεται έναν κόμβο r. Στη συνάρτηση αρχικοποιείται ο κόμβος child με τον δείκτη του αριστερού παιδιού του κόμβου r και ο δείκτης T με τον δείκτη του δεξιού παιδιού του κόμβου child. Έπειτα, το r γίνεται δεξί παιδί του κόμβου child γίνεται r και το T αριστερό παιδί του κόμβου r. Η συνάρτηση επιστρέφει τη νέα ρίζα child.

Η συνάρτηση insert τύπου void με όρισμα ένα string, η οποία θέτει στην ρίζα (root) το αποτέλεσμα της συνάρτησης insert (που θα αναφερθεί παρακάτω) με ορίσματα την ρίζα και το string.

Η συνάρτηση insert τύπου node* που δέχεται σαν ορίσματα έναν κόμβο r και ένα string. Αρχικά, αναζητά την λέξη στο δέντρο. Στη συνέχεια, ελέγχει αν ο κόμβος r είναι nullptr. Αν είναι, τότε δημιουργεί έναν νέο κόμβο στον οποίο τοποθετεί την τιμή string και τον επιστρέφει. Έπειτα, ελέγχεται αν η λέξη υπάρχει ήδη μέσα στο δένδρο. Αν δε υπάρχει, τότε ελέγχεται αν η λέξη είναι μικρότερη ή μεγαλύτερη από την τιμή του κόμβου r και αν είναι ξανακαλείται η συνάρτηση insert με όρισμα το αριστερό ή το δεξί παιδί αντίστοιχα του κόμβου r και το string. Αν η λέξη υπάρχει ήδη στο δένδρο τότε αυξάνεται η μεταβλητή count της λέξης κατά 1 και επιστρέφει το r. Ακολούθως, υπολογίζεται το Balance Factor του κόμβου r. Αν το Balance Factor του κόμβου r είναι μεγαλύτερο από το 1 ή μικρότερο από το -1 , τότε αυτό σημαίνει ότι το δένδρο είναι μη ισοζυγισμένο και χρειάζεται να γίνει περιστροφή του δένδρου, ανάλογα με την περίπτωση που ισχύει (left left case, right right case, left right case, right left case). Η συνάρτηση insert επιστρέφει τον κόμβο r.

Η λογική συνάρτηση Delete, η οποία δέχεται σαν όρισμα μια λέξη. Αρχικά, εξετάζει αν η ρίζα του δένδρου δεν είναι nullptr και αν η λέξη υπάρχει στο δένδρο. Αν οι προϋποθέσεις αυτές πληρούνται, τότε καλείται η συνάρτηση Delete (θα γίνει αναφορά για τη συνάρτηση παρακάτω) με ορίσματα την ρίζα και το string, της οποίας το αποτέλεσμα αποθηκεύεται στην ρίζα και στη συνέχεια επιστρέφει true και αν δεν ισχύουν οι προϋποθέσεις επιστρέφει false.

Η συνάρτηση Delete τύπου node* και ορίσματα έναν κόμβο r και ένα string. Αρχικά, στη συνάρτηση ελέγχεται αν η λέξη είναι μεγαλύτερη ή μικρότερη της τιμής του κόμβου r. Αν ισχύει κάποια από τις δύο περιπτώσεις, τότε ξανακαλείται η συνάρτηση Delete με όρισμα το δεξί ή το αριστερό παιδί του κόμβου r αντίστοιχα και το string. Σε περίπτωση που η λέξη είναι ίδια με την τιμή του κόμβου r , τότε διακρίνουμε τις εξής περιπτώσεις:

1. Ο κόμβος να μην έχει παιδιά. Σε αυτή την περίπτωση διαγράφεται ο κόμβος r και η συνάρτηση επιστρέφει nullptr.
2. Ο κόμβος να έχει 1 παιδί. Σε αυτή τη συνθήκη, αποθηκεύεται ο δείκτης του παιδιού σε μια node* μεταβλητή t , διαγράφεται ο κόμβος r και η συνάρτηση επιστρέφει t.
3. Ο κόμβος να έχει 2 παιδιά. Εδώ, αναζητείται στο δεξί υποδένδρο του κόμβου r ο κόμβος με την μικρότερη τιμή, που αντικαθιστά την τιμή του κόμβου r και στη συνέχεια διαγράφεται ο κόμβος του δεξιού υποδένδρου του r που έχει τιμή ίδια με την νέα τιμή του r.

Οι συναρτήσεις Search, PrintInOrder, PrintPreOrder, PrintPostOrder χρησιμοποιούν τις αντίστοιχες συναρτήσεις που κληρονομήθηκαν από το Binary Search Tree.

F. Πίνακας Κατακερματισμού (Hash table)

Ο πίνακας κατακερματισμού είναι η πιο γρήγορη δομή, τόσο από άποψη χρόνων εισαγωγής, όσο και αναζήτησης. Παρότι η αναζήτηση σε πίνακα κατακερματισμού έχει πολυπλοκότητα χειρότερης περίπτωσης O(n), λόγω της φύσης της δομής φτάνουμε υπερβολικά σπάνια στην χειρότερη περίπτωση. Σε ένα πίνακα κατακερματισμού υπάρχει μια «σωστή» θέση για κάθε στοιχείο που καθορίζεται από το «κλειδί» του HashKey το οποίο δημιουργείται από την συνάρτηση getHashKey(), η οποία προσπαθεί να δημιουργήσει όσο το δυνατόν περισσότερα διαφορετικά κλειδιά ώστε να μειώνεται ο αριθμός των συγκρούσεων, των περιπτώσεων δηλαδή που η «σωστή» θέση 2 στοιχείων είναι η ίδια.

```
long long int getHashKey(string str)
{
    long long int key=0;
    for(int i=0;i<str.length();i++)
        key+=str[i]*31*19*7*3*(i+3);
    return key;
}
```

Για να δημιουργηθεί το κλειδί προσθέτουμε κάθε φορά την αριθμητική τιμή κάθε γράμματος της λέξης επι μερικούς πρώτους αριθμούς που βοηθούν στην εξασφάλιση της σχετικής μοναδικότητας μέσω της πράξης mod που γίνεται μετά στην εισαγωγή καθώς και επι έναν όρο που εξαρτάται από τη θέση του γράμματος στην λέξη έτσι ώστε λέξεις που περιέχουν τα ίδια γράμματα με διαφορετική σειρά να παράγουν διαφορετικό hash key.

Στις private μεταβλητές τις κλάσεις υπάρχει ένας adata pointer καθώς και 2 int μεταβλητές size και words (η πρώτη το μέγιστο μέγεθος του πίνακα και η δεύτερη το πόσες λέξεις περιέχει).

Στις public υπάρχουν οι κατασκευαστές (κενός που ορίζει το size 10000 και με int όρισμα το οποίο εκχωρείται στο size) και οι συναρτήσεις Insert, Search,getCount,getWords και Print.

Στους κατασκευαστές αρχικοποιείται η τιμή (value) όλων των στοιχείων στην συμβολοσειρά «.» έτσι ώστε μετά να γίνουν οι απαραίτητες συγκρίσεις για την εισαγωγή και την αναζήτηση.

Στην Insert αφού πρώτα βρούμε το hash key της λέξης που θέλουμε να εισάγουμε, ελέγχουμε μέσω της Search αν η λέξη υπάρχει ήδη στον πίνακα κατακερματισμού. Αν υπάρχει απλά αυξάνουμε τον count της κατά 1, αν δεν υπάρχει ελέγχουμε τις θέσεις από την key mod size και μετά μέχρι να βρούμε κενή θέση (θέση της οποίας το value είναι «.» ώστε να τοποθετήσουμε εκεί την καινούργια λέξη. Στο τέλος της εισαγωγής αν η λέξη δεν υπήρχε προηγούμενη φορά, ορίζουμε τον count της ίσο με 1 και αυξάνουμε κατά 1 την μεταβλητή words. Αν οι λέξεις (τιμή της words) γίνουν παραπάνω ή ίσες με το μισό του μεγέθους, γίνεται rehashing. Δημιουργούμε δηλαδή έναν νέο πίνακα κατακερματισμού με διπλάσιο μέγεθος και εισάγουμε ξανά όσα στοιχεία από τον προηγούμενο πίνακα είχαν value διάφορο του «.» και αντιγράφουμε και κατευθείαν τον υπάρχοντα count τους. Στο τέλος δίνουμε το περιεχόμενο του νέου πίνακα στον παλιό και διπλασιάζουμε το μέγιστο μέγεθός του (size).

Στην Search ξεκινάμε από την θέση key % size που είναι και η σωστή και μέχρι να βρούμε κενή θέση (με τιμή «.») ή να βρούμε ότι υπάρχει ήδη η λέξη και να την «πετύχουμε» ξανά πηγαίνουμε κάθε φορά μία θέση δεξιά. Αν βρούμε την λέξη επιστρέφουμε την θέση της, αλλιώς επιστρέφουμε -1.

Η Print, getCount και getWords λειτουργούν όπως και στις υπόλοιπες δομές (πλην της getWords στα δέντρα).

Συμπεράσματα και αποτελέσματα.

Μέσα από την διαδικασία της δημιουργίας των δομών σχεδόν από το «μηδέν» μπορέσαμε να κατανοήσουμε πολύ περισσότερο το πώς λειτουργούν και γιατί κάποιες δομές είναι πιο γρήγορες και άλλες πιο αργές, στοιχεία που «κρύβονται» τόσο στον τρόπο υλοποίησης τους, όσο και στην φύση αυτών. Μετά και από τους ελέγχους των οποίων τα αποτελέσματα βρίσκονται παρακάτω είναι κατανοητό ότι η πιο γρήγορη δομή για αναζητήσεις είναι ο πίνακας κατακερματισμού ενώ η πιο αργή είναι ο αταξινόμητος. Οι υπόλοιπες (SortedArray, BST, AVL Tree) έχουν μικρές

διαφορές μεταξύ τους. Οι διαφορές φαίνονται πιο εύκολα στα πιο μεγάλα σύνολα.

Τα αποτελέσματα της μέτρησης χρόνου ήταν τα εξής:

Σε τυχαίο σύνολο 1000 λέξεων:

```
The search of the unsorted array took 212 milliseconds.
The search of the sorted array took 4 milliseconds.
The search of the binary search tree took 4 milliseconds.
The search of the AVL tree took 5 milliseconds.
The search of the hashtable took 3 milliseconds.
```

Σε τυχαίο σύνολο 100000 λέξεων:

```
The search of the unsorted array took 21919 milliseconds.
The search of the sorted array took 400 milliseconds.
The search of the binary search tree took 405 milliseconds.
The search of the AVL tree took 403 milliseconds.
The search of the hashtable took 312 milliseconds.
```

Σε τυχαίο σύνολο 500000 λέξεων:

```
The search of the unsorted array took 106915 milliseconds.
The search of the sorted array took 2014 milliseconds.
The search of the binary search tree took 2033 milliseconds.
The search of the AVL tree took 2005 milliseconds.
The search of the hashtable took 1587 milliseconds.
```

Συνεπώς από τα τεστ γίνεται και πάλι εμφανές το ποια είναι η πιο γρήγορη (Hash Table) και ποια είναι η πιο αργή δομή (Unsorted Array).

Για την υλοποίηση της εργασίας συμβουλευτήκαμε σελίδες στο διαδίκτυο (κυρίως GeeksForGeeks), τις διαφάνειες του μαθήματος και βίντεο στο διαδίκτυο. Το IDE που χρησιμοποιήσαμε ήταν το CLion της JetBrains.

Ομάδα:

Λιαροκάπης Αντώνιος-Παναγιώτης ΑΕΜ: 3932

Μαρέδης Ιωάννης ΑΕΜ: 3931