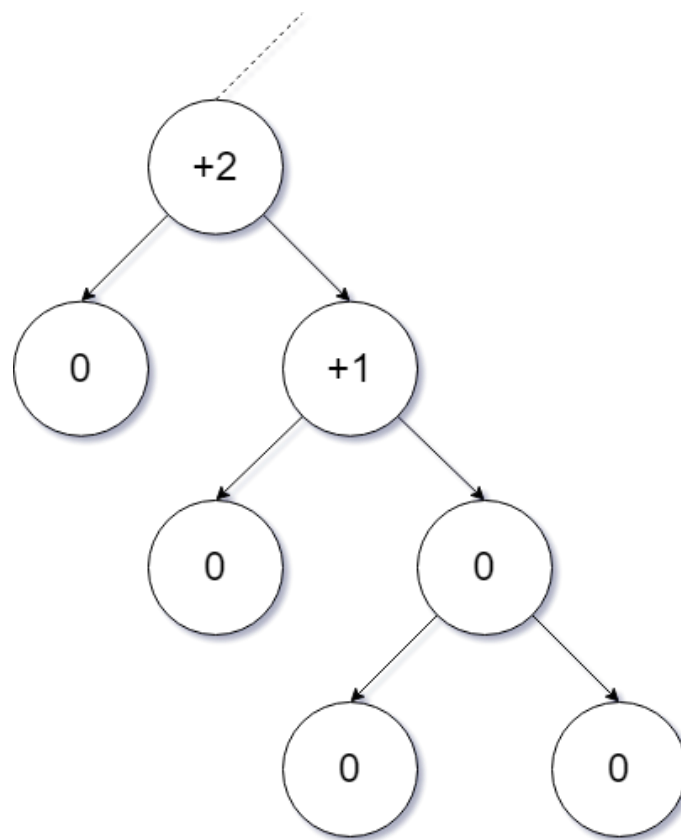


ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ



ΔΗΜΗΤΡΙΟΣ ΦΑΝΑΡΙΩΤΗΣ

1084597

ΝΙΚΟΛΑΟΣ ΑΝΔΡΙΑΝΟΠΟΥΛΟΣ

1084637

ΠΑΝΑΓΙΩΤΗΣ ΚΑΛΟΖΟΥΜΗΣ

1084560

ΣΟΦΙΑ ΚΑΝΑΚΗ

1084655

Περιεχόμενα

ΜΕΡΟΣ Ι – ΑΝΑΖΗΤΗΣΗ ΚΑΙ ΤΑΞΙΝΟΜΗΣΗ

1. Εισαγωγή

1.1	Αναπαράσταση των δεδομένων.....	7
1.2	Αποθήκευση των δεδομένων.....	8
1.2.1	Η συνάρτηση fileLinesSize	9
1.2.2	Η συνάρτηση readLine.....	10
1.2.3	Η συνάρτηση makeData	10
1.2.4	Η συνάρτηση readCsv	12
1.2.5	Η συνάρτηση exportToCsv.....	13
1.3	Τρόπος εκτέλεση πειραματικών συγκρίσεων	14

2. Ταξινόμηση

2.1	Insertion Sort και Quick Sort.....	16
2.1.1	Insertion Sort	16
2.1.1.1	Η συνάρτηση insertionSort.....	16
2.1.2	Quick Sort	16
2.1.2.1	Η συνάρτηση quicksort.....	17
2.1.2.2	Η συνάρτηση swap	17
2.1.3	Πειραματικές συγκρίσεις και συμπεράσματα.....	17
2.2	Heap Sort και Counting Sort.....	18

2.2.1	Heap Sort	18
2.2.1.1	Η συνάρτηση heapSort	19
2.2.1.2	Η συνάρτηση heapify	19
2.2.2	Counting Sort	20
2.2.2.1	Η συνάρτηση findRange	20
2.2.2.2	Η συνάρτηση countingSort	21
2.2.3	Πειραματικές συγκρίσεις και συμπεράσματα.....	21

3. Αναζήτηση

3.1	Προεπεξεργασία των δεδομένων	23
3.1.1	Η συνάρτηση compareDates	24
3.1.2	Η συνάρτηση sortDates	25
3.2	Δυαδική Αναζήτηση vs Αναζήτηση με Παρεμβολή	25
3.2.1	Δυαδική Αναζήτηση	26
3.2.1.1	Η συνάρτηση binSearch.....	26
3.2.2	Αναζήτηση με Παρεμβολή	29
3.2.2.1	Η συνάρτηση dateDifference	30
3.2.2.2	Η συνάρτηση isLeapYear	31
3.2.2.3	Η συνάρτηση interSearch	32
3.2.3	Πειραματικές συγκρίσεις και συμπεράσματα.....	35
3.3	Δυαδική Αναζήτηση Παρεμβολής και η παραλλαγή της	38
3.3.1	Δυαδική Αναζήτηση Παρεμβολής.....	38
3.3.1.1	Η συνάρτηση sqroot.....	38
3.3.1.2	Η συνάρτηση bis	39
3.3.2	Παραλλαγή.....	40
3.3.2.1	Η συνάρτηση binpow	40
3.3.2.2	Η συνάρτηση altbis.....	41
3.3.3	Πειραματικές συγκρίσεις και συμπεράσματα.....	41

ΜΕΡΟΣ II – ΔΕΝΤΡΑ AVL ΚΑΙ HASHING

4. Δέντρα AVL

4.1	Αναπαράσταση των δεδομένων	43
4.2	Αποθήκευση των δεδομένων	45
4.2.1	Η συνάρτηση readDateandTemp	46
4.2.2	Η συνάρτηση freeTree	48
4.3	Περιστροφές	49
4.3.1	Απλή περιστροφή.....	49
4.3.1.1	Η συνάρτηση leftRotation.....	49
4.3.1.1	Η συνάρτηση rightRotation	52
4.3.2	Διπλή περιστροφή	55
4.3.2.1	Η συνάρτηση leftRightRotation.....	55
4.3.2.2	Η συνάρτηση rightLeftRotation	56
4.4	Βασικές πράξεις πάνω σε δέντρα AVL	56
4.4.1	Αναζήτηση.....	57
4.4.1.1	Η συνάρτηση dateSearch	57
4.4.1.1	Η συνάρτηση tempSearch.....	58
4.4.2	Ένθεση.....	58
4.4.2.1	Η συνάρτηση insertDateAVLnode.....	59
4.4.2.2	Η συνάρτηση insertTempAVLnode	61
4.4.3	Διαγραφή.....	63
4.4.3.1	Η συνάρτηση deleteAVLnode.....	63
4.5	Υπολογισμός ύψους και υψοζύγισης.....	65
4.5.1	Η συνάρτηση balanceFinder	65
4.5.2	Η συνάρτηση deleteRotation	68
4.5.3	Η συνάρτηση max	71
4.6	Άλλες συναρτήσεις	71
4.6.1	Η συνάρτηση inOrder.....	72
4.6.2	Η συνάρτηση minTemp	73
4.6.3	Η συνάρτηση maxTemp	73

5. Κατακερματισμός

5.1	Αναπαράσταση των δεδομένων	75
5.2	Συνάρτηση κατακερματισμού	76
5.2.1	Η συνάρτηση getKey	76
5.2.2	Η συνάρτηση hash	77
5.3	Αποθήκευση των δεδομένων	77
5.3.1	Η συνάρτηση insert	78
5.3.2	Η συνάρτηση insertAll	78
5.4	Υλοποίηση πράξεων	78
5.4.1	Η συνάρτηση searchNode	79
5.4.2	Η συνάρτηση editNode	79
5.4.3	Η συνάρτηση deleteNode	80

6. Το τελικό πρόγραμμα

6.1	Συναρτήσεις του μενού	81
6.1.1	Η συνάρτηση menuOption	81
6.1.2	Η συνάρτηση inputDate	82
6.2	Σχεδίαση του μενού	82
6.3	Λειτουργίες	84

Κεφάλαιο 1

Εισαγωγή

Στο κεφάλαιο αυτό αναλύουμε πώς από το αρχείο **ocean.csv** μετατρέπουμε τα δεδομένα μας σε μια μορφή που θα μας επιτρέπει να τα επεξεργαζόμαστε μέσα στο πρόγραμμά μας και να εφαρμόζουμε διάφορες πράξεις πάνω σ' αυτά, όπως ταξινόμηση και αναζήτηση.

Ενότητα 1.1 Αναπαράσταση των δεδομένων

Το αρχείο **ocean.csv** αποτελείται από γραμμές, όπου κάθε γραμμή-δείγμα αντιστοιχεί σε μια μέτρηση της θερμοκρασίας (σε °C) στο νερό του ωκεανού καθώς και των τιμών Phosphate, Silicate, Nitrite, Nitrate, Salinity, Oxygen για την περίοδο 2000 έως 2019. Η δομή κάθε γραμμής είναι:

Date	Temp	Phosphate	Silicate	Nitrite	Nitrate	Salinity	Oxygen
------	------	-----------	----------	---------	---------	----------	--------

Εικόνα 1.1 Δομή μιας γραμμής του ocean.csv

Για την αποθήκευση κάθε μιας απ' αυτές τις γραμμές χρησιμοποιούμε ένα δικό μας struct με όνομα **Data** το οποίο διαθέτει τα παρακάτω πεδία:

```
typedef struct Data
{
    Date date;
    float values[7];
}
Data;
```

Το πεδίο **date** είναι επίσης ένα δικό μας struct που μας επιτρέπει να διαχειριζόμαστε με ευκολία τις ημερομηνίες που εμφανίζονται στο πρόγραμμά μας. Διαθέτει τα παρακάτω πεδία:

```
typedef struct Date
{
    int day;
    int month;
    int year;
}
Date;
```

Το πεδίο **values** είναι ένας πίνακας 7 πραγματικών, όπου κάθε θέση του αντιστοιχεί σε μια από τις τιμές Temp έως Oxygen που φαίνονται στο Σχήμα 1.1. Για ευκολία στην ανάγνωση, αντιστοιχίσαμε στον προεπεξεργαστή κάθε μια από τις θέσεις αυτού του πίνακα σε μια σταθερά, όπως φαίνεται παρακάτω:

```
#define TEMP values[0]
#define PHOSPHATE values[1]
#define SILICATE values[2]
#define NITRITE values[3]
#define NITRATE values[4]
#define SALINITY values[5]
#define OXYGEN values[6]
```

Το αρχείο **ocean.csv** αποτελείται από πολλά τέτοια δείγματα. Επομένως, για να αποθηκεύσουμε όλη την πληροφορία του αρχείου στο πρόγραμμά μας, απαιτείται ένας πίνακας από **Data**, όπου κάθε θέση του θα αντιστοιχεί και σε ένα δείγμα:

```
Data* dataArray;
```

Ο παραπάνω πίνακας θα περιέχει όλη την πληροφορία του αρχείου **ocean.csv**, καθώς το κελί **dataArray[i]** αντιστοιχεί στην (i+2)-οστή γραμμή του αρχείου, ή αλλιώς στο (i+1)-οστό δείγμα του αρχείου.

Ενότητα 1.2 Αποθήκευση των δεδομένων

Αφού είδαμε πώς μπορούμε να αναπαραστήσουμε τα δεδομένα μας μέσα στο πρόγραμμα, ας δούμε τώρα πώς γίνεται η μετατροπή των δεδομένων από τη μορφή που περιέχονται στο αρχείο στη μορφή που περιγράψαμε προηγουμένως.

Στόχος μας είναι αφού διαβάζουμε μια γραμμή του αρχείου, να μπορούμε έπειτα να τη χωρίζουμε στα επιμέρους κομμάτια της και να τα αποθηκεύουμε σε ένα struct τύπου **Data**. Επαναλαμβάνοντας αυτή τη διαδικασία για κάθε γραμμή του **ocean.csv**, δημιουργούμε τελικά τον ζητούμενο πίνακα **dataArray** που περιέχει όλη την πληροφορία του αρχείου.

Όλες οι απαιτούμενες ενέργειες για την ανάγνωση του αρχείου και τη δημιουργία του πίνακα επιτυγχάνονται με συναρτήσεις που θα αναλύσουμε στη συνέχεια.

1.2.1 Η συνάρτηση `fileLinesSize`

Δήλωση:

```
int* fileLinesSize(char* filename);
```

Παράμετροι:

- **filename** – Το όνομα του csv αρχείου που πρόκειται να διαβαστεί.

Επιστρέφει:

Έναν πίνακα 2 ακεραίων όπου το πρώτο κελί είναι το πλήθος των δειγμάτων μέσα στο αρχείο και το δεύτερο κελί είναι το μέγιστο πλήθος χαρακτήρων σε μια γραμμή.

Η συνάρτηση **fileLinesSize** διαβάζει το αρχείο που δηλώνεται από το όρισμα **filename** και επιστρέφει έναν δυναμικά δεσμευμένο πίνακα 2 θέσεων, έστω με το όνομα **array**, όπου στη θέση **array[0]** περιέχεται το πλήθος των δειγμάτων που περιέχονται μέσα σ' αυτό και στη θέση **array[1]** το μέγιστο πλήθος χαρακτήρων σε μια γραμμή, λαμβάνοντας υπ' όψη και τον χαρακτήρα αλλαγής γραμμής. Αυτές οι δύο τιμές μας βοηθούν να προσδιορίσουμε τον απαραίτητο χώρο στη μνήμη που πρέπει να δεσμευτεί ώστε να αποθηκευτεί σωστά η πληροφορία του αρχείου. Συγκεκριμένα:

- Το **array[0]** μας δίνει το μέγεθος του πίνακα **dataArray** που θα χρειαστεί να δεσμεύσουμε στη μνήμη.
- Το **array[1]** μας δίνει το μέγεθος του προσωρινού αλφαριθμητικού όπου θα αποθηκευτεί μια γραμμή του αρχείου που διαβάζουμε, προτού αυτή χωριστεί στα επιμέρους κομμάτια της.

Για να λειτουργήσει σωστά η συνάρτηση, το .csv αρχείο από το οποίο γίνεται η ανάγνωση πρέπει να βρίσκεται στην αρχική του μορφή, δηλαδή να περιέχει μια γραμμή με άχρηστη πληροφορία στην αρχή και ακριβώς μια κενή γραμμή στο τέλος. Αυτό συμβαίνει επειδή στην πραγματικότητα αυτό που υπολογίζουμε είναι το πλήθος των γραμμών του αρχείου μειωμένο κατά 2 και όχι το πλήθος των γραμμών με χρήσιμη πληροφορία.

1.2.2 Η συνάρτηση `readLine`

Δήλωση:

```
char* readLine(FILE* file, int linesize);
```

Παράμετροι:

- **file** – Δείκτης προς το csv αρχείο από το οποίο γίνεται η ανάγνωση.
- **linesize** – Το μέγιστο πλήθος χαρακτήρων σε μια γραμμή του αρχείου file.

Επιστρέφει:

Έναν πίνακα με όλους τους χαρακτήρες της επόμενης προς ανάγνωση γραμμής.

Η συνάρτηση **readLine** διαβάζει τους χαρακτήρες μιας γραμμής του αρχείου **file** μέχρι το τέλος της (δηλαδή μέχρι να συναντήσει τον χαρακτήρα αλλαγής γραμμής) ή μέχρι να φτάσει το μέγιστο πλήθος χαρακτήρων που καθορίζεται από το όρισμα **linesize**. Το από που θα ξεκινήσει η ανάγνωση της γραμμής καθορίζεται από έναν ειδικό δείκτη θέσης του αρχείου, ο οποίος μετακινείται μετά από κάθε ανάγνωση. Επομένως, επανειλημμένες κλήσεις αυτής της συνάρτησης πάνω στο ίδιο αρχείο εγγυώνται ότι κάθε φορά θα διαβάζουμε το αρχείο γραμμή-γραμμή.

1.2.3 Η συνάρτηση `makeData`

Δήλωση:

```
Data makeData(char* line);
```

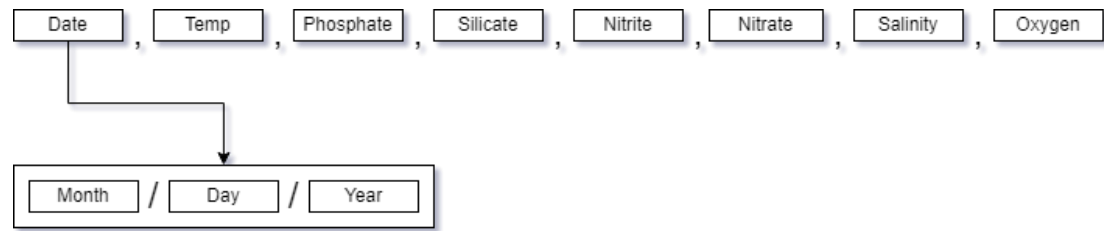
Παράμετροι:

- **line** – Πίνακας χαρακτήρων που αντιστοιχεί σε μια γραμμή του αρχείου.

Επιστρέφει:

Ένα struct τύπου **Data** που περιέχει όλη την πληροφορία της γραμμής **line**

Η συνάρτηση **makeData** δέχεται ως όρισμα μια γραμμή του αρχείου **ocean.csv** που διαβάστηκε προηγουμένως και την χωρίζει στα επιμέρους κομμάτια της, αποθηκεύοντάς τα τελικά σε ένα struct τύπου **Data**. Χρησιμοποιεί τη συνάρτηση **strtok** της C, η οποία χωρίζει ένα αλφαριθμητικό σε κομμάτια λαμβάνοντας υπ' όψη κάποιον διαχωριστή που εμείς ορίζουμε. Η δομή μιας γραμμής του αρχείου μαζί με τους διαχωριστές φαίνεται στο παρακάτω σχήμα:



Εικόνα 1.2 Διαχωριστές μιας γραμμής του αρχείου ocean.csv

Παρατηρούμε ότι κάθε γραμμή αποτελείται από 8 τιμές οι οποίες διαχωρίζονται με το κόμμα (.). Επομένως, σε αρχικό στάδιο απομονώνουμε αυτές τις τιμές. Την ημερομηνία την αποθηκεύουμε σε ένα ξεχωριστό αλφαριθμητικό ώστε να τη διαχειριστούμε στη συνέχεια, ενώ όλες τις άλλες τιμές τις μετατρέπουμε από αλφαριθμητικά σε πραγματικούς με τη συνάρτηση **atof** και τις αποθηκεύουμε σε ένα νέο struct τύπου **Data**:

Data:

```

Data newData;
int i = 0;
char* dateString = (char*)malloc(sizeof(char));

char* token = strtok(line, ",");
while(token)
{
    if (i == 0)
    {
        //The first token is the date, which is stored in dateString
        //for processing later
        dateString = (char*)realloc(dateString,
        (strlen(token)+1)*sizeof(char));
        strcpy(dateString, token);
    }
    else newData.values[i-1] = atof(token);

    token = strtok(NULL, ",");
    i++;
}

```

Αφού αποθηκεύσουμε όλες τις άλλες τιμές, στο τέλος διαχωρίζουμε ξεχωριστά το αλφαριθμητικό **dateString**, το οποίο χρησιμοποιεί ως διαχωριστή την κάθετο (/). Τα επιμέρους κομμάτια αποθηκεύονται σε ένα struct τύπου **Date** το οποίο τελικά αποθηκεύεται στο **Data** που θέλουμε να δημιουργήσουμε.

```

i = 0;

//Breaking the date string into tokens, this time using "/" as a
//delimiter
token = strtok(dateString, "/");
while(token)
{
    if (i == 0)
        newData.date.month = atoi(token);
    else if (i == 1)
        newData.date.day = atoi(token);
}

```

```

else newData.date.year = atoi(token);

token = strtok(NULL, "/");
i++;
}

```

Μετά από αυτή τη διαδικασία, το **Data** μας για τη γραμμή **line** έχει δημιουργηθεί και μπορεί να επιστραφεί από τη συνάρτηση:

```

free(dateString);
return newData;

```

1.2.4 Η συνάρτηση readCsv

Δήλωση:

```
Data* readCsv(char* filename, int maxsize, int linesize)
```

Παράμετροι:

- **filename** – Το όνομα του csv αρχείου που πρόκειται να διαβαστεί.
- **maxsize** – Το πλήθος των δειγμάτων μέσα στο αρχείο filename.
- **linesize** – Το μέγιστο πλήθος χαρακτήρων σε μια γραμμή του αρχείου filename.

Επιστρέφει:

Έναν πίνακα με όλη την πληροφορία του αρχείου filename.

Η συνάρτηση **readCsv** ανοίγει και διαβάζει το .csv αρχείο που δίνεται από το όρισμα **filename** και επιστρέφει έναν πίνακα από **Data** που κάθε θέση του αντιστοιχεί σε μια γραμμή του αρχείου. Για τις τιμές των ορισμάτων **maxsize** και **linesize** πρέπει από πριν να έχει κληθεί ξεχωριστά η συνάρτηση **fileLinesize**.

Η συνάρτηση αυτή αρχικά δεσμεύει στη μνήμη έναν πίνακα από **Data** μεγέθους **maxsize**, δηλαδή όσα και τα δείγματα του αρχείου:

```
array = (Data*) malloc(maxsize * sizeof(Data));
```

Για να αρχίσει επαναληπτικά η ανάγνωση του αρχείου, πρέπει πρώτα να καλέσουμε μια φορά τη συνάρτηση **readLine** χωρίς να κρατάμε κάπου το αποτέλεσμα, ώστε να μετακινηθεί ο δείκτης του αρχείου από την πρώτη γραμμή που περιέχει άχρηστη πληροφορία στην αρχή της επόμενης γραμμής. Δεν ξεχνάμε φυσικά να αποδεσμεύσουμε τη μνήμη που δεσμεύσαμε για την άχρηστη γραμμή που διαβάσαμε:

```

//Get rid of the first line before reading the other ones
free(readLine(newFile, linesize));

```

Έπειτα μπορούμε με μια επαναληπτική διαδικασία να διαβάζουμε γραμμές από το αρχείο, να δημιουργούμε το αντίστοιχο **Data** και τελικά να το αποθηκεύουμε στην αντίστοιχη θέση του τελικού πίνακα:

```
for (int i = 0; i < maxsize; i++)
{
    char* s2 = readLine(newFile, linesize);
    array[i] = makeData(s2);
    free(s2);
}
```

Για να λειτουργήσει σωστά η συνάρτηση, το .csv αρχείο από το οποίο γίνεται η ανάγνωση πρέπει να βρίσκεται στην αρχική του μορφή, δηλαδή να περιέχει μια γραμμή με άχρηστη πληροφορία στην αρχή και ακριβώς μια κενή γραμμή στο τέλος:

- Η έλλειψη της πρώτης, άχρηστης γραμμής του αρχείου οδηγεί στην αγνόηση της πρώτης μέτρησης, καθώς πάντα ξεφορτωνόμαστε την πρώτη γραμμή του αρχείου.
- Η έλλειψη της τελευταίας, κενής γραμμής του αρχείου οδηγεί στην αγνόηση της τελευταίας μέτρησης.
- Η ύπαρξη παραπάνω κενών γραμμών στο τέλος του αρχείου οδηγεί σε σφάλμα.

1.2.5 Η συνάρτηση exportToCsv

Δήλωση:

```
void exportToCsv(const Data* data, char* filename, int maxsize)
```

Παράμετροι:

- **data** – Ο πίνακας με όλη την πληροφορία που θέλουμε να αποθηκεύσουμε.
- **filename** – Το όνομα του αρχείου που πρόκειται να δημιουργηθεί.
- **maxsize** – Το μέγεθος του πίνακα data.

Η συνάρτηση **exportToCsv** επιτελεί την αντίστροφη λειτουργία της συνάρτησης **readCsv**. Δημιουργεί ένα νέο αρχείο με όνομα **filename** και αποθηκεύει σε κάθε γραμμή του την πληροφορία της αντίστοιχης θέσης του πίνακα **data**, σε παρόμοια μορφή με το πρωτότυπο αρχείο .csv. Στην πρώτη θέση του νέου αρχείου γράφεται, όπως και στο πρωτότυπο η παρακάτω γραμμή:

Date,T_degC,P04uM,SiO3uM,N02uM,N03uM,Salnty,O2ml_L

```

Date, T_degC, PO4uM, SiO3uM, NO2uM, NO3uM, Salnty, O2ml_L
01/07/2000, 11, 2.23, 53, 0, 25.3, 33.895, 2.11
01/08/2000, 11.21, 2.91, 73.3, 0, 28.3, 34.022, 0.28
01/09/2000, 17.95, 0.19, 3, 0, 0, 33.622, 5.64
01/10/2000, 17.91, 0.19, 3, 0, 0, 33.621, 5.66
01/11/2000, 17.55, 0.18, 3, 0, 0, 33.616, 5.83
01/12/2000, 17.18, 0.16, 3, 0, 0, 33.612, 5.89
01/13/2000, 16.57, 0.24, 4, 0, 0, 33.61, 5.89
01/14/2000, 15.43, 0.3, 4.4, 0.02, 0.2, 33.619, 5.67
01/15/2000, 14.77, 0.4, 6, 0.05, 0.7, 33.627, 5.43
01/16/2000, 13.78, 0.54, 10.5, 0.02, 3.7, 33.685, 4.94
01/17/2000, 12.23, 1.18, 22.6, 0, 12, 33.788, 3.93
01/18/2000, 11.61, 1.69, 37.4, 0, 20.3, 33.888, 2.93
01/19/2000, 17.73, 0.29, 5, 0, 0.1, 33.63, 5.74
01/20/2000, 17.05, 0.29, 4, 0, 0.1, 33.63, 5.77
01/21/2000, 13.38, 0.4, 7, 0.05, 1.8, 33.57, 5.89
01/22/2000, 11.34, 0.88, 16, 0.06, 9, 33.59, 5.1
01/23/2000, 9.1, 2.06, 58, 0, 27.6, 34, 2.71
04/07/2000, 8.71, 2.16, 73, 0, 30.5, 34.14, 2.53

```

Date	T_degC	PO4uM	SiO3uM	NO2uM	NO3uM	Salnty	O2ml_L
01/07/2000	11.000	02.230	53.000	00.000	25.300	33.895	02.110
01/08/2000	11.210	02.910	73.300	00.000	28.300	34.022	00.280
01/09/2000	17.950	00.190	03.000	00.000	00.000	33.622	05.640
01/10/2000	17.910	00.190	03.000	00.000	00.000	33.621	05.660
01/11/2000	17.550	00.180	03.000	00.000	00.000	33.616	05.830
01/12/2000	17.180	00.160	03.000	00.000	00.000	33.612	05.890
01/13/2000	16.570	00.240	04.000	00.000	00.000	33.610	05.890
01/14/2000	15.430	00.300	04.400	00.020	00.200	33.619	05.670
01/15/2000	14.770	00.400	06.000	00.050	00.700	33.627	05.430
01/16/2000	13.780	00.540	10.500	00.020	03.700	33.685	04.940
01/17/2000	12.230	01.180	22.600	00.000	12.000	33.788	03.930
01/18/2000	11.610	01.690	37.400	00.000	20.300	33.888	02.930
01/19/2000	17.730	00.290	05.000	00.000	00.100	33.630	05.740
01/20/2000	17.050	00.290	04.000	00.000	00.100	33.630	05.770
01/21/2000	13.380	00.400	07.000	00.050	01.800	33.570	05.890
01/22/2000	11.340	00.880	16.000	00.060	09.000	33.590	05.100
01/23/2000	09.100	02.060	58.000	00.000	27.600	34.000	02.710
04/07/2000	08.710	02.160	73.000	00.000	30.500	34.140	02.530

Εικόνα 1.3 Το πρωτότυπο ocean.csv αρχείο και το νέο αρχείο που δημιουργεί η exportToCsv

Η συνάρτηση αυτή θα είναι ιδιαίτερα χρήσιμη αργότερα όταν μετά την ταξινόμηση των αρχικών δεδομένων θέλουμε να τα αποθηκεύσουμε πάλι σε ένα αρχείο .csv.

Ενότητα 1.3 Τρόπος εκτέλεσης πειραματικών συγκρίσεων

Στα προγράμματά μας εκτελούμε πολλούς διαφορετικούς αλγόριθμους για λειτουργίες όπως η αναζήτηση και η ταξινόμηση. Θέλουμε να μπορούμε να συγκρίνουμε πειραματικά τους χρόνους εκτέλεσης των αλγορίθμων αυτών. Σε αυτή την ενότητα θα δούμε τη διαδικασία που ακολουθούμε για την πειραματική σύγκριση των αλγορίθμων.

Για τον υπολογισμό του χρόνου εκτέλεσης χρησιμοποιούμε τη συνάρτηση **clock**, την οποία καλούμε δύο φορές, πριν και μετά την αρχική κλήση κάθε αλγορίθμου στην **main**, ώστε να υπολογίζουμε τη διαφορά στον χρόνο:

```

clock_t t1 = clock();

(κώδικας του οποίου τον χρόνο εκτέλεσης μετράμε)

t1 = clock() - t1;

```

Η μεταβλητή **t1** περιέχει το πλήθος των κύκλων ρολογιού που χρειάστηκε η συνάρτηση. Για να υπολογίσουμε τον χρόνο εκτέλεσης σε δευτερόλεπτα, διαιρούμε με τη σταθερά **CLOCKS_PER_SEC**:

```
(double) t1 / CLOCKS_PER_SEC
```

Εδώ πρέπει να επισημάνουμε ότι επειδή οι αλγόριθμοι εκτελούνται πολύ γρήγορα στους υπολογιστές μας, με αποτέλεσμα οι χρόνοι να προκύπτουν μηδενικοί, χρειάστηκε για το πειραματικό κομμάτι να εκτελέσουμε τους αλγορίθμους σε ένα online περιβάλλον, το οποίο εκτέλεσε το πρόγραμμα με μεγαλύτερη καθυστέρηση και μας επέτρεπε να εξάγουμε συμπεράσματα σχετικά με τον χρόνο εκτέλεσης. Το περιβάλλον που χρησιμοποιήθηκε μπορεί να βρεθεί στον παρακάτω σύνδεσμο:

<https://www.jdoodle.com/c-online-compiler/>

Κεφάλαιο 2

Ταξινόμηση

Στο κεφάλαιο αυτό εξετάζουμε διάφορους αλγορίθμους ταξινόμησης.

Ενότητα 2.1 Insertion Sort και Quick Sort

Αρχικά θα δούμε τον Insertion Sort και τον Quick Sort, τους οποίους χρησιμοποιούμε για να ταξινομήσουμε τις μετρήσεις με βάση τις θερμοκρασίες.

2.1.1 Insertion Sort

Ο αλγόριθμος Insertion Sort υλοποιείται στο πρόγραμμά μας χρησιμοποιώντας τη συνάρτηση `insertionSort`, την οποία εξετάζουμε παρακάτω.

2.1.1.1 Η συνάρτηση `insertionSort`

Δήλωση:

```
void insertionSort(Data arr[], int size)
```

Παράμετροι:

- **arr[]** – Ο πίνακας με τα δεδομένα.
- **size** – Το μέγεθος του πίνακα arr.

Η συνάρτηση `insertionSort` χρησιμοποιεί τον αλγόριθμο του Insertion Sort για να ταξινομήσει κατά αύξουσα σειρά τα δεδομένα του πίνακα ως προς τη θερμοκρασία. Αξίζει να σημειωθεί πως στον έλεγχο για την ανταλλαγή των δεδομένων, αν τα δεδομένα έχουν την ίδια θερμοκρασία τότε συγκρίνουμε τις ημερομηνίες και μετακινούμε πιο αριστερά το δεδομένο με την μικρότερη ημερομηνία.

2.1.2 Quick Sort

Ο αλγόριθμος Quick Sort υλοποιείται στο πρόγραμμά μας αναδρομικά χρησιμοποιώντας τη συνάρτηση `quicksort`, την οποία αναλύουμε στη συνέχεια.

2.1.2.1 Η συνάρτηση quicksort

Δήλωση:

```
void quicksort(Data arr[], int low, int high)
```

Παράμετροι:

- **arr[]** – Ο πίνακας με τα δεδομένα.
- **low** – Το κάτω όριο του πίνακα.
- **high** – Το πάνω όριο του πίνακα.

Η συνάρτηση **quicksort** χρησιμοποιεί τον αλγόριθμο του Quick Sort για να ταξινομήσει κατά αύξουσα σειρά τα δεδομένα του πίνακα ως προς τη θερμοκρασία. Αξίζει να σημειωθεί πως στον έλεγχο για την ανταλλαγή των δεδομένων, αν τα δεδομένα έχουν την ίδια θερμοκρασία τότε συγκρίνουμε τις ημερομηνίες και μετακινούμε πιο αριστερά το δεδομένο με την μικρότερη ημερομηνία.

Η ανταλλαγή των δεδομένων γίνεται με τη χρήση της συνάρτησης **swap**.

2.1.2.2 Η συνάρτηση swap

Δήλωση:

```
void swap(Data *a, Data *b)
```

Παράμετροι:

- **a** – Δείκτης προς το πρώτο Data.
- **b** – Δείκτης προς το δεύτερο Data.

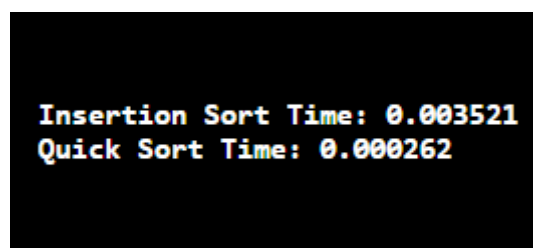
Η συνάρτηση **swap** δέχεται ως ορίσματα δύο δείκτες προς structs τύπου **Data** και αντιμεταθέτει τα δεδομένα τους. Χρησιμοποιείται στη συνάρτηση **quicksort** όταν θέλουμε να κάνουμε αντιμετάθεση δύο θέσεων.

2.1.3 Πειραματικές συγκρίσεις και συμπεράσματα

Τώρα που είδαμε πώς λειτουργούν και οι δύο αλγόριθμοι αναζήτησης, ας συγκρίνουμε τα αποτελέσματά τους.

Αρχικά, γνωρίζουμε από τη θεωρία ότι στη χειρότερη περίπτωση ο αλγόριθμος Insertion Sort εκτελείται σε χρόνο $O(n^2)$, ενώ ο αλγόριθμος Quick Sort έχει χρόνο χειρότερης περίπτωσης $O(n^2)$, αλλά στη μέση περίπτωση παίρνει χρόνο $O(n \log n)$, ο οποίος είναι καλύτερος από τον χρόνο χειρότερης περίπτωσης του Insertion Sort. Περιμένουμε, επομένως, ο Quick Sort να είναι πιο αποδοτικός από τον Insertion Sort. Στη συνέχεια θα εκτελέσουμε τους δύο αλγορίθμους και θα εξάγουμε συμπεράσματα σχετικά με τον χρόνο εκτέλεσης. Για τον υπολογισμό του χρόνου εκτέλεσης, ακολουθούμε τη διαδικασία που περιγράψαμε στην Ενότητα 1.3.

Τρέχοντας τους δύο αλγορίθμους, παίρνουμε τα παρακάτω αποτελέσματα:



```
Insertion Sort Time: 0.003521
Quick Sort Time: 0.000262
```

Εικόνα 2.1 Χρόνος εκτέλεσης των αλγορίθμων Insertion Sort και Quick Sort

Παρατηρούμε ότι ο Insertion Sort εκτελείται σε 3521 μ s, ενώ ο Quick Sort σε μόνο 262 μ s. Επιβεβαιώνουμε ότι πράγματι ο Quick Sort είναι γρηγορότερος.

Ενότητα 2.2 Heap Sort και Counting Sort

Στη συνέχεια θα δούμε τους αλγορίθμους Heap Sort και τον Counting Sort, τους οποίους χρησιμοποιούμε για να ταξινομήσουμε τις μετρήσεις με βάση τις τιμές phosphate.

2.2.1 Heap Sort

Ο αλγόριθμος Heap Sort υλοποιείται στο πρόγραμμά μας χρησιμοποιώντας τη συνάρτηση **heapSort**, την οποία αναλύουμε στη συνέχεια. Ο σωρός που χρησιμοποιεί ο αλγόριθμος κατασκευάζεται με τη συνάρτηση **heapify**.

2.2.1.1 Η συνάρτηση heapSort

Δήλωση:

```
void heapSort(Data arr[], int size)
```

Παράμετροι:

- **arr[]** – Ο πίνακας με τα δεδομένα.
- **size** – Το μέγεθος του πίνακα arr.

Η συνάρτηση **heapSort** χρησιμοποιεί τον αλγόριθμο του Heap Sort για να ταξινομήσει κατά αύξουσα σειρά τα δεδομένα του πίνακα ως προς τις τιμές phosphate.

Ο αλγόριθμος χωρίζεται σε δύο φάσεις. Στην πρώτη φάση, ξεκινάμε από τη θέση **father = size/2** και για **father > 0**, κατασκευάζουμε έναν σωρό μεγίστων με τη διαδικασία **heapify**. Μόλις ο σωρός έχει κατασκευαστεί, για **father = 0**, ξεκινά η δεύτερη φάση του αλγορίθμου, που είναι η ταξινόμηση κατά αύξουσα σειρά. Αξίζει να σημειωθεί πως στον έλεγχο για την ανταλλαγή των δεδομένων, αν τα δεδομένα έχουν την ίδια θερμοκρασία τότε συγκρίνουμε τις ημερομηνίες και μετακινούμε πιο αριστερά το δεδομένο με την μικρότερη ημερομηνία.

Ο αλγόριθμος Heap Sort έχει υλοποιηθεί με τέτοιον τρόπο ώστε να είναι επιτόπιος, δηλαδή δε δεσμεύει επιπλέον χώρο για την παραγωγή του ταξινομημένου πίνακα, αλλά χρησιμοποιεί τον πίνακα εισόδου. Αυτό επιτυγχάνεται αντιμετωπίζοντας κάθε φορά το πρώτο στοιχείο του σωρού με το τελευταίο, εφαρμόζοντας τη διαδικασία **heapify** για το νέο πρώτο στοιχείο για να αποκατασταθεί η ιδιότητα του σωρού και μειώνοντας το μέγεθος του σωρού κάθε φορά κατά 1.

2.2.1.2 Η συνάρτηση heapify

Δήλωση:

```
void heapify(Data arr[], int size, int tmpFather)
```

Παράμετροι:

- **arr[]** – Ο πίνακας με τα δεδομένα.
- **size** – Ο μέγιστος δείκτης του πίνακα arr.
- **tmpFather** – Ο κόμβος τον οποίο θα μετακινήσουμε κάτω.

Η συνάρτηση **heapify** χρησιμοποιείται για την κατασκευή του σωρού μεγίστων που απαιτείται από τον αλγόριθμο Heap Sort. Είναι ουσιαστικά ο αλγόριθμος heapify down,

ο οποίος σε συνδυασμό με την πρώτη φάση της συνάρτησης **heapSort**, κατασκευάζει έναν σωρό σε μόλις γραμμικό χρόνο, αξιοποιώντας το γεγονός ότι ένας οποιοσδήποτε πίνακας είναι σωρός από τη θέση **size/2**.

Η συνάρτηση αυτή εξετάζει ποιο από τα δύο παιδιά του **tmpFather** έχει τη μέγιστη τιμή phosphate και το αντιμετωπίζει με τον πατέρα. Αν πατέρας και παιδί έχουν την ίδια τιμή phosphate, πατέρας γίνεται αυτός με τη μεγαλύτερη ημερομηνία.

2.2.2 Counting Sort

Ο αλγόριθμος Counting Sort υλοποιείται στο πρόγραμμά μας χρησιμοποιώντας τη συνάρτηση **countingSort**. Θέλουμε να χρησιμοποιήσουμε τον αλγόριθμο για να ταξινομήσουμε τις τιμές phosphate. Ωστόσο, ο αλγόριθμος δουλεύει μόνο με ακέραιες τιμές και τα phosphate είναι δεκαδικές. Στην απεικόνιση των δεκαδικών σε ακραίους μας βοηθάει η συνάρτηση **findRange**.

2.2.2.1 Η συνάρτηση findRange

Δήλωση:

```
int* findRange(Data arr[],int size)
```

Παράμετροι:

- **arr[]** – Ο πίνακας με τα δεδομένα.
- **size** – Το μέγεθος του πίνακα arr.

Επιστρέφει:

Έναν πίνακα 2 ακεραίων όπου το πρώτο κελί είναι η ελάχιστη ακέραια τιμή και το δεύτερο κελί είναι το εύρος των ακεραίων.

Η συνάρτηση **findRange** δέχεται ως όρισμα έναν πίνακα από **Data** και βρίσκει την ελάχιστη και τη μέγιστη τιμή απ' όλα τα phosphate. Στη συνέχεια απεικονίζει αυτές τις τιμές στους ακεραίους πολλαπλασιάζοντας με το 100 και υπολογίζει το εύρος μεταξύ μέγιστης και ελάχιστης τιμής (στο σύνολο των ακεραίων). Επιστρέφει έναν δυναμικά δεσμευμένο πίνακα 2 θέσεων, όπου:

- Το **array[0]** περιέχει την ελάχιστη τιμή phosphate απεικονισμένη στους ακεραίους.
- Το **array[1]** περιέχει το εύρος των ακεραίων, ώστε να ξέρουμε το μέγεθος του πίνακα.

2.2.2.2 Η συνάρτηση countingSort

Δήλωση:

```
Data* countingSort (Data arr[], int size)
```

Παράμετροι:

- **arr []** – Ο πίνακας με τα δεδομένα.
- **size** – Το μέγεθος του πίνακα arr.

Επιστρέφει:

Τον ταξινομημένο πίνακα.

Η συνάρτηση **countingSort** χρησιμοποιεί τον αλγόριθμο του Counting Sort για να ταξινομήσει κατά αύξουσα σειρά τα δεδομένα του πίνακα ως προς τις τιμές phosphate. Χρησιμοποιεί τη συνάρτηση **findRange** για να βρει τόσο τον ελάχιστο ακέραιο, όσο και το μέγεθος του συνόλου των ακεραίων. Έπειτα εφαρμόζει τον αλγόριθμο όπως και στην περίπτωση των ακεραίων, με μερικές αλλαγές π.χ. **index=arr[i] . PHOSPHATE*100**, για να μπορούμε να δουλέψουμε στο πεδίο των ακεραίων και να ενημερώνουμε σωστά τον πίνακα με τους counters των τιμών.

Σε αντίθεση με τους προηγούμενους αλγορίθμους ταξινόμησης που εξετάζουμε, ο συγκεκριμένος αλγόριθμος δεν κάνει κάτι στην περίπτωση όπου τα phosphate έχουν την ίδια τιμή.

Μόλις ολοκληρωθεί ο αλγόριθμος, ο δυναμικά δεσμευμένος ταξινομημένος πίνακας επιστρέφεται από τη συνάρτηση.

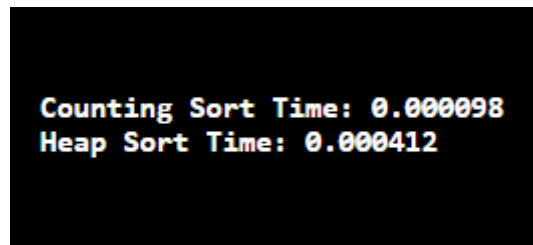
2.2.3 Πειραματικές συγκρίσεις και συμπεράσματα

Τώρα που είδαμε πώς λειτουργούν και οι δύο αλγόριθμοι αναζήτησης, ας συγκρίνουμε τα αποτελέσματά τους.

Αρχικά, γνωρίζουμε από τη θεωρία ότι στη χειρότερη περίπτωση ο αλγόριθμος Heap Sort εκτελείται σε χρόνο **$O(n \log n)$** , ενώ ο αλγόριθμος Counting Sort έχει χρόνο χειρότερης περίπτωσης **$O(n+k)$** , όπου n το μέγεθος του πίνακα ακεραίων και $[1...k]$ είναι το διάστημα στο οποίο βρίσκονται οι ακέραιοι. Περιμένουμε, επομένως, ο Counting Sort να είναι πιο αποδοτικός από τον Heap Sort. Στη συνέχεια θα εκτελέσουμε τους δύο αλγορίθμους και θα εξάγουμε συμπεράσματα σχετικά με τον χρόνο εκτέλεσης. Για τον

υπολογισμό του χρόνου εκτέλεσης, ακολουθούμε τη διαδικασία που περιγράψαμε στην Ενότητα 1.3.

Τρέχοντας τους δύο αλγορίθμους, παίρνουμε τα παρακάτω αποτελέσματα:



```
Counting Sort Time: 0.000098
Heap Sort Time: 0.000412
```

Εικόνα 2.2 Χρόνος εκτέλεσης των αλγορίθμων Heap Sort και Counting Sort

Παρατηρούμε ότι ο Heap Sort εκτελείται σε 412 μ s, ενώ ο Counting Sort σε μόνο 98 μ s. Επιβεβαιώνουμε ότι πράγματι ο Counting Sort είναι γρηγορότερος.

Κεφάλαιο 3

Αναζήτηση

Στο κεφάλαιο αυτό εξετάζουμε τους αλγορίθμους της δυαδικής αναζήτησης, της αναζήτησης με παρεμβολή και της δυαδικής αναζήτησης με παρεμβολή (μαζί με μία παραλλαγή της) και συγκρίνουμε πειραματικά τα αποτελέσματά τους.

Ενότητα 3.1 Προεπεξεργασία των δεδομένων

Οι αλγόριθμοι αναζήτησης που θα υλοποιήσουμε σε αυτό το κεφάλαιο βασίζονται στο γεγονός ότι το σύνολο των στοιχείων μας είναι ήδη ταξινομημένο, πετυχαίνοντας έτσι καλύτερους χρόνους χειρότερης περίπτωσης από αυτόν της γραμμικής αναζήτησης. Συγκεκριμένα, εφόσον θέλουμε να αναζητάμε πληροφορίες χρησιμοποιώντας μια ημερομηνία, πρέπει τα δεδομένα μας να είναι ταξινομημένα ως προς τις ημερομηνίες. Ωστόσο, με μια προσεκτική παρατήρηση του αρχείου **ocean.csv** διαπιστώνουμε εύκολα ότι:

```
03/14/2009,13.73,0.36,2,0.02,0.4,33.21,6.13
03/14/2009,13.28,0.47,2,0.06,1.5,33.24,6.02
03/15/2009,9.29,1.55,24,0.01,21.8,33.67,3.86
03/15/2009,8.58,1.93,35,0,28.2,33.94,3.04
03/16/2009,15.2,0.33,2,0,0,33.15,5.33
03/16/2009,15.18,0.28,2,0,0,33.15,6.12
03/17/2009,14.65,0.28,2,0,0,33.16,6.12
03/17/2009,13.91,0.33,2,0.02,0.3,33.24,6.06
03/13/2009,9.39,1.1,16,0.01,14.5,33.55,4.68
03/12/2009,8.5,1.68,32,0.01,25.7,33.88,3.56
03/11/2009,16.25,0.3,2,0,0,33.15,5.7
03/08/2009,15.68,0.28,2,0,0,33.11,5.71
03/09/2009,14.89,0.3,2,0,0,33.18,5.84
03/10/2009,10.12,0.83,11,0.01,9,33.4,5.11
```

Εικόνα 3.1 Μη ταξινομημένα δεδομένα στο αρχείο ocean.csv

Το αρχείο μας δεν είναι πάντα ταξινομημένο. Παρ' όλα αυτά, υπάρχουν μεγάλα τμήματα του αρχείου τα οποία είναι πλήρως ταξινομημένα:

```

01/07/2000,11,2.23,53,0,25.3,33.895,2.11
01/08/2000,11.21,2.91,73.3,0,28.3,34.022,0.28
01/09/2000,17.95,0.19,3,0,0,33.622,5.64
01/10/2000,17.91,0.19,3,0,0,33.621,5.66
01/11/2000,17.55,0.18,3,0,0,33.616,5.83
01/12/2000,17.18,0.16,3,0,0,33.612,5.89
01/13/2000,16.57,0.24,4,0,0,33.61,5.89
01/14/2000,15.43,0.3,4.4,0.02,0.2,33.619,5.67
01/15/2000,14.77,0.4,6,0.05,0.7,33.627,5.43
01/16/2000,13.78,0.54,10.5,0.02,3.7,33.685,4.94
01/17/2000,12.23,1.18,22.6,0,12,33.788,3.93
01/18/2000,11.61,1.69,37.4,0,20.3,33.888,2.93
01/19/2000,17.73,0.29,5,0,0.1,33.63,5.74
01/20/2000,17.05,0.29,4,0,0.1,33.63,5.77

```

Εικόνα 3.2 Ταξινομημένα δεδομένα στο αρχείο ocean.csv

Σε μια τέτοια περίπτωση, όπου τα δεδομένα μας είναι σχεδόν ταξινομημένα, ο πιο αποδοτικός αλγόριθμος ταξινόμησης είναι ο **Insertion Sort**, καθώς για μεγάλο τμήμα των ημερομηνιών δε θα χρειαστεί καν να γίνει κάποια αντιμετάθεση. Θα δούμε στη συνέχεια τις τροποποιήσεις που πρέπει να κάνουμε στον **Insertion Sort** που υλοποιήσαμε στο Κεφάλαιο 2 ώστε να ταξινομεί το αρχείο ως προς τις ημερομηνίες.

Πριν μπορέσουμε να ταξινομήσουμε τις ημερομηνίες, χρειάζεται να ορίσουμε μια διάταξη πάνω στο σύνολο των ημερομηνιών. Προφανώς, η διάταξη αυτή έχει να κάνει με το ποια ημερομηνία είναι παλαιότερη της άλλης π.χ. ισχύει ότι $25/3/2022 < 28/10/2022$. Αυτή η σύγκριση γίνεται με τη συνάρτηση **compareDates** την οποία θα εξετάσουμε στη συνέχεια.

3.1.1 Η συνάρτηση compareDates

Δήλωση:

```
int compareDates(const Date *d1, const Date *d2)
```

Παράμετροι:

- **d1** – Η πρώτη ημερομηνία που συμμετέχει στη σύγκριση.
- **d2** – Η δεύτερη ημερομηνία που συμμετέχει στη σύγκριση.

Επιστρέφει:

Έναν ακέραιο που δείχνει το αποτέλεσμα της σύγκρισης

Η συνάρτηση **compareDates** δέχεται ως ορίσματα δύο structs τύπου **Date** (συγκεκριμένα τους δείκτες τους, ώστε να μη δημιουργούνται νέα structs), τα **d1** και **d2** και επιστρέφει το αποτέλεσμα της σύγκρισής τους. Συγκεκριμένα:

- Αν **d1 > d2** επιστρέφεται 1
- Αν **d1 = d2** επιστρέφεται 0
- Αν **d1 < d2** επιστρέφεται -1

3.1.2 Η συνάρτηση sortDates

Δήλωση:

```
void sortDates(struct Data* data, int maxsize)
```

Παράμετροι:

- **data** – Ο πίνακας που πρόκειται να ταξινομηθεί.
- **maxsize** – Το μέγεθος του πίνακα **data**.

Η συνάρτηση **sortDates** ταξινομεί τον πίνακα **data** που περιέχει όλη την πληροφορία του αρχείου **ocean.csv** με βάση της ημερομηνίες χρησιμοποιώντας τον αλγόριθμο **Insertion Sort**, ο οποίος είναι ιδιαίτερα αποδοτικός για ταξινόμηση σχεδόν ταξινομημένων πινάκων.

Η υλοποίηση αυτής της συνάρτησης είναι παρόμοια με αυτήν του **Insertion Sort** στο Κεφάλαιο 2, με τη μόνη διαφορά ότι όπου πριν συγκρίναμε θερμοκρασίες, τώρα συγκρίνουμε ημερομηνίες χρησιμοποιώντας τη συνάρτηση **compareDates**.

Ενότητα 3.2 Δυαδική αναζήτηση vs Αναζήτηση με Παρεμβολή

Τώρα που έχουμε ορίσει μια διάταξη πάνω στις ημερομηνίες και καταφέραμε τα ταξινομήσουμε τα αρχικά μας δεδομένα, μπορούμε να αρχίσουμε να εκτελούμε αλγορίθμους αναζήτησης πάνω σ' αυτά. Σε αυτήν την ενότητα εξετάζουμε τους δύο πρώτους αλγορίθμους που αναφέραμε, τη δυαδική αναζήτηση και την αναζήτηση με παρεμβολή. Αφού υλοποιήσουμε αυτούς τους αλγορίθμους, στο τέλος της ενότητας τούς συγκρίνουμε πειραματικά μετρώντας τον χρόνο εκτέλεσης και τον αριθμό των βημάτων που κάθε ένας απ' αυτούς απαιτεί.

3.2.1 Δυαδική Αναζήτηση

Η δυαδική αναζήτηση στο πρόγραμμά μας υλοποιείται αναδρομικά χρησιμοποιώντας τη συνάρτηση `binSearch`, η οποία αναλύεται στη συνέχεια.

3.2.1.1 Η συνάρτηση `binSearch`

Δήλωση:

```
int binSearch(const Data* data, int* left, int* right, const Date*
date, const int maxIndex, int* steps)
```

Παράμετροι:

- **data** – Ο πίνακας στον οποίο θα γίνει η δυαδική αναζήτηση.
- **left** – Δείκτης προς το αριστερό όριο του τμήματος του πίνακα στο οποίο εφαρμόζουμε τη δυαδική αναζήτηση.
- **right** – Δείκτης προς το δεξιό όριο του τμήματος του πίνακα στο οποίο εφαρμόζουμε τη δυαδική αναζήτηση.
- **date** – Η ημερομηνία που αναζητάμε.
- **maxIndex** – Ο μέγιστος αριθμός θέσης όλου του πίνακα.
- **steps** – Ο αριθμός των βημάτων (κλήσεων) που χρειάζεται η αναζήτηση μέχρι την εύρεση του **date**.

Επιστρέφει:

Έναν ακέραιο που δείχνει αν το στοιχείο βρέθηκε ή όχι.

Η συνάρτηση `binSearch` εφαρμόζει τον αλγόριθμο της δυαδικής αναζήτησης πάνω στα δεδομένα του ήδη ταξινομημένου πίνακα **data**, αναζητώντας την ημερομηνία που καθορίζεται από το όρισμα **date**.

Όταν καλούμε τη συνάρτηση για πρώτη φορά, ορίζουμε στην **main** δύο μεταβλητές **left** και **right** με αρχικές τιμές 0 και **maxIndex** αντίστοιχα και περνάμε στα αντίστοιχα ορίσματα δείκτες προς τις μεταβλητές αυτές. Με κάθε κλήση της συνάρτησης, συγκρίνεται η τιμή του μεσαίου στοιχείου του πίνακα με το στοιχείο που ψάχνουμε:

```
int arrayMid = (*left + *right)/2;

switch(compareDates(date, &data[arrayMid].date))
{
    .
    .
    .
}
```

Αν η ημερομηνία που ψάχνουμε είναι μικρότερη από την ημερομηνία στη μέση (αποτέλεσμα σύγκρισης -1), τότε εστιάζουμε την προσοχή μας στο πρώτο μισό του πίνακα αλλάζοντας κατάλληλα την τιμή του **right**:

```
case -1:
{
    *right = arrayMid - 1;
    return binSearch(data, left, right, date, maxIndex, steps);
}
```

Αν η ημερομηνία που ψάχνουμε είναι μεγαλύτερη από την ημερομηνία στη μέση (αποτέλεσμα σύγκρισης 1), τότε εστιάζουμε την προσοχή μας στο δεύτερο μισό του πίνακα αλλάζοντας κατάλληλα την τιμή του **left**:

```
case 1:
{
    *left = arrayMid + 1;
    return binSearch(data, left, right, date, maxIndex, steps);
}
```

Αν η ημερομηνία που ψάχνουμε ταυτίζεται με την ημερομηνία στη μέση (αποτέλεσμα σύγκρισης 0), τότε θεωρούμε ότι βρήκαμε τη θέση του στοιχείου. Ωστόσο, επειδή ενδέχεται να υπάρχουν παραπάνω από ένα δείγματα με την ίδια ημερομηνία, αναζητάμε κι' άλλα στοιχεία με την ίδια ημερομηνία, τα οποία θα βρίσκονται γύρω από το μεσαίο στοιχείο λόγω του ότι ο πίνακας είναι ταξινομημένος. Επομένως, ξεκινώντας από τη μέση, προσδιορίζουμε τα όρια του υποπίνακα ο οποίος περιέχει εξ' ολοκλήρου τη ζητούμενη ημερομηνία. Η παράμετρος **maxIndex** χρησιμοποιείται επίσης ώστε να φροντίζουμε ότι παραμένουμε στα όρια του αρχικού πίνακα:

```
case 0:
{
    //Left and right will now become the limits of the sub-array
    where every element is the date we want

    *left = arrayMid;
    *right = arrayMid;

    while ((*left > 0) && (compareDates(date, &data[*left-1].date) ==
0))
        *left = *left - 1;

    while ((*right < maxIndex) && (compareDates(date,
&data[*right+1].date) == 0))
        *right = *right + 1;

    return 1;
}
```

Αυτός είναι και ο λόγος που περνάμε τα όρια του πίνακα ως δείκτες αντί για απλούς ακεραίους. Θέλουμε μετά την κλήση της συνάρτησης οι δύο μεταβλητές να πάρουν στη **main** τις τιμές που καθορίζουν τα όρια του υποπίνακα που βρήκαμε παραπάνω. Προφανώς αν η ημερομηνία που αναζητάμε είναι μοναδική, τότε **left = right**.

Για να δούμε ότι η συνάρτηση πράγματι βρίσκει όλες τις ίδιες ημερομηνίες, μπορούμε να την καλέσουμε για την ημερομηνία 17/3/2009, η οποία εμφανίζεται στο αρχείο **ocean.csv** 3 φορές. Παρατηρούμε ότι εκτυπώνονται οι μετρήσεις και για τα 3 δείγματα:

```
Please input a date:

Day: 17

Month: 3

Year: 2009

===== BINARY SEARCH RESULTS =====

Temperature: 14.870
Phosphate: 0.360

Temperature: 14.650
Phosphate: 0.280

Temperature: 13.910
Phosphate: 0.330

Binary Search Time: 0.000000 (8 steps)

=====
```

Εικόνα 3.3 Αναζήτηση της ημερομηνίας 17/3/2009

Στην παραπάνω εικόνα βλέπουμε ότι μαζί με τα αποτελέσματα της αναζήτησης, εκτυπώνεται και πληροφορία σχετικά με τα βήματα που χρειάστηκε η συνάρτηση για να εκτελεστεί. Όπως οι μεταβλητές **left** και **right**, έτσι και η μεταβλητή **steps** ορίζεται στην **main** και περνάει στη συνάρτηση με αναφορά, ώστε η υπολογισμένη τιμή για τα βήματα να διατηρείται και έξω απ' τη συνάρτηση. Με κάθε κλήση της συνάρτησης η τιμή αυτή αυξάνεται.

```
*steps = *steps + 1;
```

Η καταμέτρηση των βημάτων με τη μεταβλητή **steps** αποτελεί μια αναγκαστική λύση στο πρόβλημα προσδιορισμού του χρόνου εκτέλεσης, καθώς η συνάρτηση **clock**, την οποία επίσης χρησιμοποιούμε για τον ίδιο σκοπό, πολλές φορές δίνει μηδενικό αποτέλεσμα λόγω της ταχύτητας εκτέλεσης του προγράμματος.

Τέλος, αν το στοιχείο βρεθεί μέσα στον πίνακα, επιστρέφεται η τιμή 1, εναλλακτικά επιστρέφεται 0.

```
if (*left > *right || compareDates(date, &data[*left].date) == -1 ||
    compareDates(date, &data[*right].date) == 1)
{
    return 0;
}
```

3.2.2 Αναζήτηση με Παρεμβολή

Ο αλγόριθμος της αναζήτησης με παρεμβολή είναι παρόμοιος με τον αλγόριθμο της δυαδικής αναζήτησης, με τη μόνη σημαντική διαφορά ότι αντί να εξετάζουμε κάθε φορά το μεσαίο στοιχείο, εξετάζουμε το:

$$\text{left} + \left\lfloor \frac{\text{date} - \text{data}[\text{left}].\text{date}}{\text{data}[\text{right}].\text{date} - \text{data}[\text{left}].\text{date}} (\text{right} - \text{left}) \right\rfloor$$

Παρατηρούμε, επομένως, ότι πρέπει να ορίσουμε μια σχέση η οποία αντιστοιχίζει δύο ημερομηνίες σε έναν ακέραιο. Πιο συγκεκριμένα, χρειαζόμαστε έναν τρόπο να βρίσκουμε τη διαφορά δύο ημερομηνιών σε μέρες. Αυτό το καταφέρνουμε με τη συνάρτηση **dateDifference** την οποία εξετάζουμε στη συνέχεια.

Αφού ορίσουμε την παραπάνω συνάρτηση, μπορούμε να αρχίσουμε να υλοποιούμε την αναζήτηση με παρεμβολή. Η αναζήτηση με παρεμβολή στο πρόγραμμά μας υλοποιείται αναδρομικά χρησιμοποιώντας τη συνάρτηση **interSearch**, την οποία επίσης θα δούμε στη συνέχεια.

3.2.2.1 Η συνάρτηση dateDifference

Αήλωση:

```
int dateDifference(const Date* endDate, const Date* startDate)
```

Παράμετροι:

- **endDate** – Η αφαιρετέα ημερομηνία.
- **startDate** – Η ημερομηνία-αφαιρέτης.

Επιστρέφει:

Τη διαφορά των δύο ημερομηνιών σε μέρες

Η συνάρτηση **dateSearch** υπολογίζει και επιστρέφει τη διαφορά **endDate** – **startDate** σε μέρες. Ο υπολογισμός χωρίζεται σε 3 ξεχωριστά μέρη.

Πρώτα υπολογίζεται η διαφορά μεταξύ της τελικής ημερομηνίας και της πρώτης μέρας του έτους της τελικής ημερομηνίας:

```
//Adding the days since the start of the year
days = endDate->day;
for (int i = 1; i <= endDate->month-1; i++)
{
    days += daysOf[endDate->month - i - 1];
    if (endDate->month - i - 1 == 1)
    {
        days += isLeapYear(endDate->year);
    }
}
```

Έπειτα προστίθενται όλες οι μέρες των ετών από την αρχική μέχρι την τελική ημερομηνία:

```
//Adding the days until we reach the start year
for (int i = 1; i < yearDif; i++)
{
    days += 365 + isLeapYear(endDate->year - i);
}
```

Τέλος, υπολογίζεται η διαφορά μεταξύ της τελευταίας ημέρας του έτους της αρχικής ημερομηνίας και της αρχικής ημερομηνίας:

```
//Calculating 31/12/startDate->year - startDate
```

```
days += 31 - startDate->day;
for (int i = 1; i <= 12 - startDate->month; i++)
{
    days += daysOf[11 - i];
    if (11 - i == 1)
    {
        days += isLeapYear(startDate->year);
    }
}
```

Για την περίπτωση όπου η τελική και η αρχική χρονιά ταυτίζονται, αρκεί μόνο μια γενίκευση του τελευταίου βήματος:

```
days = endDate->day - startDate->day;
for (int i = 1; i <= monthDif; i++)
{
    days += daysOf[endDate->month - i - 1];
    if (endDate->month - i - 1 == 1)
    {
        days += isLeapYear(endDate->year);
    }
}
```

Άλλες λεπτομέρειες σχετικά με τη συνάρτηση:

- Για τις ανάγκες της αναζήτησης με παρεμβολή, θεωρήσαμε μόνο την περίπτωση όπου πάντα **endDate** \geq **startDate**.
- Η τελική μέρα δε συμπεριλαμβάνεται στον υπολογισμό π.χ. 30/5/2022 - 29/5/2022 = 1 μέρα.
- Λαμβάνουμε υπ' όψη μας τα δίσεκτα έτη. Ελέγχουμε αν ένα έτος είναι δίσεκτο με τη συνάρτηση **isLeapYear** την οποία εξετάζουμε στη συνέχεια.

3.2.2.2 Η συνάρτηση `isLeapYear`

Δήλωση:

```
int isLeapYear(int year)
```

Παράμετροι:

- **year** – Το έτος που θέλουμε να ελέγξουμε αν είναι δίσεκτο.

Επιστρέφει:

Έναν ακέραιο που δείχνει αν το έτος **year** είναι δίσεκτο ή όχι.

Η συνάρτηση **isLeapYear** δέχεται ως όρισμα έναν ακέραιο που αντιστοιχεί σε ένα έτος και επιστρέφει 1 αν αυτό είναι δίσεκτο, εναλλακτικά 0. Επιστρέφει, δηλαδή το αποτέλεσμα της παρακάτω λογικής έκφρασης:

```
return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
```

3.2.2.3 Η συνάρτηση interSearch

Δήλωση:

```
int interSearch(const Data* data, int* left, int* right, const Date* date, const int maxIndex, int* steps)
```

Παράμετροι:

- **data** – Ο πίνακας στον οποίο θα γίνει η αναζήτηση με παρεμβολή.
- **left** – Δείκτης προς το αριστερό όριο του τμήματος του πίνακα στο οποίο εφαρμόζουμε την αναζήτηση με παρεμβολή.
- **right** – Δείκτης προς το δεξιό όριο του τμήματος του πίνακα στο οποίο εφαρμόζουμε την αναζήτηση με παρεμβολή.
- **date** – Η ημερομηνία που αναζητάμε.
- **maxIndex** – Ο μέγιστος αριθμός θέσης όλου του πίνακα.
- **steps** – Ο αριθμός των βημάτων (κλήσεων) που χρειάζεται η αναζήτηση μέχρι την εύρεση του **date**.

Επιστρέφει:

Έναν ακέραιο που δείχνει αν το στοιχείο βρέθηκε ή όχι.

Η συνάρτηση **interSearch** εφαρμόζει τον αλγόριθμο της αναζήτησης με παρεμβολή πάνω στα δεδομένα του ήδη ταξινομημένου πίνακα **data**, αναζητώντας την ημερομηνία που καθορίζεται από το όρισμα **date**.

Όταν καλούμε τη συνάρτηση για πρώτη φορά, ορίζουμε στην **main** δύο μεταβλητές **left** και **right** με αρχικές τιμές 0 και **maxIndex** αντίστοιχα και περνάμε στα αντίστοιχα ορίσματα δείκτες προς τις μεταβλητές αυτές. Με κάθε κλήση της συνάρτησης, συγκρίνουμε το στοιχείο που ψάχνουμε με το στοιχείο στη θέση **next**, η οποία δίνεται από την παρακάτω σχέση:

$$\text{left} + \left\lfloor \frac{\text{date} - \text{data}[\text{left}].\text{date}}{\text{data}[\text{right}].\text{date} - \text{data}[\text{left}].\text{date}} (\text{right} - \text{left}) \right\rfloor$$

Επομένως:

```
int next = ((double) (dateDifference (date,
&data[*left].date)) / (dateDifference (&data[*right].date,
&data[*left].date))) * (*right-*left) + *left;

switch (compareDates (date, &data[next].date))
{
    .
    .
    .
}
```

Αν η ημερομηνία που ψάχνουμε είναι μικρότερη από την ημερομηνία στο **next** (αποτέλεσμα σύγκρισης -1), τότε εστιάζουμε την προσοχή μας στο πρώτο κομμάτι του πίνακα αλλάζοντας κατάλληλα την τιμή του **right**:

```
case -1:
{
    *right = next - 1;
    return interSearch (data, left, right, date, maxIndex, steps);
}
```

Αν η ημερομηνία που ψάχνουμε είναι μεγαλύτερη από την ημερομηνία στο **next** (αποτέλεσμα σύγκρισης 1), τότε εστιάζουμε την προσοχή μας στο δεύτερο κομμάτι του πίνακα αλλάζοντας κατάλληλα την τιμή του **left**:

```
case 1:
{
    *left = next + 1;
    return interSearch (data, left, right, date, maxIndex, steps);
}
```

Αν η ημερομηνία που ψάχνουμε ταυτίζεται με την ημερομηνία στο **next** (αποτέλεσμα σύγκρισης 0), τότε θεωρούμε ότι βρήκαμε τη θέση του στοιχείου. Ωστόσο, επειδή ενδέχεται να υπάρχουν παραπάνω από ένα δείγματα με την ίδια ημερομηνία, αναζητάμε κι' άλλα στοιχεία με την ίδια ημερομηνία, τα οποία θα βρίσκονται γύρω από το **next** λόγω του ότι ο πίνακας είναι ταξινομημένος. Επομένως, ξεκινώντας από το **next**, προσδιορίζουμε τα όρια του υποπίνακα ο οποίος περιέχει εξ' ολοκλήρου τη ζητούμενη ημερομηνία. Η παράμετρος **maxIndex** χρησιμοποιείται επίσης ώστε να φροντίζουμε ότι παραμένουμε στα όρια του αρχικού πίνακα:

```
case 0:
{
```

```

    //Left and right will now become the limits of the sub-array
    where every element is the date we want

    *left = next;
    *right = next;

    while ((*left > 0) && (compareDates(date, &data[*left-1].date) ==
0))
        *left = *left - 1;

    while ((*right < maxIndex) && (compareDates(date,
&data[*right+1].date) == 0))
        *right = *right + 1;

    return 1;
}

```

Αυτός είναι και ο λόγος που περνάμε τα όρια του πίνακα ως δείκτες αντί για απλούς ακεραίους. Θέλουμε μετά την κλήση της συνάρτησης οι δύο μεταβλητές να πάρουν στη **main** τις τιμές που καθορίζουν τα όρια του υποπίνακα που βρήκαμε παραπάνω. Προφανώς αν η ημερομηνία που αναζητάμε είναι μοναδική, τότε **left = right**.

Για να δούμε ότι η συνάρτηση πράγματι βρίσκει όλες τις ίδιες ημερομηνίες, μπορούμε να την καλέσουμε για την ημερομηνία 17/3/2009, η οποία εμφανίζεται στο αρχείο **ocean.csv** 3 φορές. Παρατηρούμε ότι εκτυπώνονται οι μετρήσεις και για τα 3 δείγματα:

```

Please input a date:

Day: 17

Month: 3

Year: 2009

===== INTERPOLATION SEARCH RESULTS =====

Temperature: 14.870
Phosphate: 0.360

Temperature: 14.650
Phosphate: 0.280

Temperature: 13.910
Phosphate: 0.330

Interpolation Search Time: 0.000000 (9 steps)

=====

```

Εικόνα 3.3 Αναζήτηση της ημερομηνίας 17/3/2009

Στην παραπάνω εικόνα βλέπουμε ότι μαζί με τα αποτελέσματα της αναζήτησης, εκτυπώνεται και πληροφορία σχετικά με τα βήματα που χρειάστηκε η συνάρτηση για να εκτελεστεί. Όπως οι μεταβλητές **left** και **right**, έτσι και η μεταβλητή **steps** ορίζεται στην **main** και περνάει στη συνάρτηση με αναφορά, ώστε η υπολογισμένη τιμή για τα βήματα να διατηρείται και έξω απ' τη συνάρτηση. Με κάθε κλήση της συνάρτησης η τιμή αυτή αυξάνεται.

```
*steps = *steps + 1;
```

Η καταμέτρηση των βημάτων με τη μεταβλητή **steps** αποτελεί μια αναγκαστική λύση στο πρόβλημα προσδιορισμού του χρόνου εκτέλεσης, καθώς η συνάρτηση **clock**, την οποία επίσης χρησιμοποιούμε για τον ίδιο σκοπό, πολλές φορές δίνει μηδενικό αποτέλεσμα λόγω της ταχύτητας εκτέλεσης του προγράμματος.

Τέλος, αν το στοιχείο βρεθεί μέσα στον πίνακα, επιστρέφεται η τιμή 1, εναλλακτικά επιστρέφεται 0.

```
if (*left > *right || compareDates(date, &data[*left].date) == -1 ||  
compareDates(date, &data[*right].date) == 1)  
{  
    return 0;  
}
```

3.2.3 Πειραματικές συγκρίσεις και συμπεράσματα

Τώρα που είδαμε πώς λειτουργούν και οι δύο αλγόριθμοι αναζήτησης, ας συγκρίνουμε τα αποτελέσματά τους.

Αρχικά, γνωρίζουμε από τη θεωρία ότι στη χειρότερη περίπτωση ο αλγόριθμος δυαδικής αναζήτησης εκτελείται σε χρόνο **$O(\log n)$** , ενώ ο αλγόριθμος αναζήτησης με παρεμβολή έχει χρόνο χειρότερης περίπτωσης **$O(n)$** και χρόνο μέσης περίπτωσης **$O(\log \log n)$** , ο οποίος είναι καλύτερος από τον χρόνο χειρότερης περίπτωσης της δυαδικής αναζήτησης. Στην επίτευξη αυτού του μέσου χρόνου, ωστόσο, χαρακτηριστικό ρόλο παίζει η κατανομή των δεδομένων. Συγκεκριμένα, όσο πιο ομοιόμορφα κατανεμημένα είναι τα δεδομένα μας, τόσο πιο αποτελεσματική είναι η αναζήτηση με παρεμβολή. Μάλιστα, εάν τα δεδομένα μας είναι πλήρως ομοιόμορφα κατανεμημένα, ο αλγόριθμος βρίσκει τη θέση του στοιχείου σε ένα μόνο βήμα! Από την άλλη, ο αλγόριθμος δυαδικής αναζήτησης δεν επηρεάζεται καθόλου από την κατανομή των δεδομένων. Ελέγχει πάντα το μεσαίο στοιχείο ανεξάρτητα κατανομής και έχει σαν μοναδική προϋπόθεση ότι τα δεδομένα μας είναι ταξινομημένα.

Στην περίπτωση μας, τα δεδομένα μας δεν είναι ομοιόμορφα κατανομημένα για όλον τον πίνακα, αλλά για μέρη του. Περιμένουμε, επομένως, η αναζήτηση με παρεμβολή να είναι σε αρκετές περιπτώσεις πιο αποδοτική από τη δυαδική αναζήτηση, αλλά όχι πάντα. Στη συνέχεια θα εκτελέσουμε τους δύο αλγορίθμους και θα εξάγουμε συμπεράσματα σχετικά με τον χρόνο εκτέλεσης ή/και τον αριθμό των βημάτων που απαιτούν. Ο αριθμός των βημάτων υπολογίζεται μέσα στις συναρτήσεις, όπως δείξαμε προηγουμένως. Για τον υπολογισμό του χρόνου εκτέλεσης, ακολουθούμε τη διαδικασία που περιγράψαμε στην Ενότητα 1.3.

Αρχικά, ας τρέξουμε τους δύο αλγορίθμους για την ημερομηνία 1/2/2002:

```
Please input a date:
Day:
Month:
Year:
===== BINARY SEARCH RESULTS =====

Temperature: 14.730
Phosphate: 0.470

Binary Search Time: 0.000005 (8 steps)

=====

===== INTERPOLATION SEARCH RESULTS =====

Temperature: 14.730
Phosphate: 0.470

Interpolation Search Time: 0.000003 (1 steps)

=====
```

Εικόνα 3.4 Εκτέλεση των αλγορίθμων για την ημερομηνία 1/2/2002

Παρατηρούμε ότι η δυαδική αναζήτηση απαιτεί χρόνο 5 μ s και εκτελείται σε 8 βήματα, ενώ η αναζήτηση με παρεμβολή απαιτεί χρόνο 3 μ s και εκτελείται σε ένα μόνο βήμα. Αυτό ήταν ένα παράδειγμα αποτελεσματικής εκτέλεσης της αναζήτησης με παρεμβολή. Για τις περισσότερες ημερομηνίες που εισάγουμε, μπορούμε να περιμένουμε παρόμοια αποτελέσματα. Ας τρέξουμε τώρα τους αλγορίθμους για μια άλλη ημερομηνία, έστω 7/4/2008:

```
Please input a date:
Day:
Month:
Year:
===== BINARY SEARCH RESULTS =====

Temperature: 8.580
Phosphate: 2.100

Temperature: 14.080
Phosphate: 0.720

Binary Search Time: 0.000002 (4 steps)

=====

===== INTERPOLATION SEARCH RESULTS =====

Temperature: 8.580
Phosphate: 2.100

Temperature: 14.080
Phosphate: 0.720

Interpolation Search Time: 0.000006 (11 steps)

=====
```

Εικόνα 3.5 Εκτέλεση των αλγορίθμων για την ημερομηνία 7/4/2008

Παρατηρούμε ότι η δυαδική αναζήτηση απαιτεί χρόνο 2 μ s και εκτελείται σε μόνο 4 βήματα, ενώ η αναζήτηση με παρεμβολή απαιτεί χρόνο 6 μ s και εκτελείται σε 11 βήματα, τα οποία είναι πολύ χειρότερα αποτελέσματα σε σχέση με πριν. Αυτό, όπως προαναφέρθηκε, οφείλεται στο ότι τα δεδομένα μας δεν είναι πλήρως ομοιόμορφα κατανομημένα. Αν όλες οι ημερομηνίες ήταν ισαπέχουσες, περιπτώσεις όπως η παραπάνω δεν θα ήταν ποτέ πρόβλημα.

Ενότητα 3.3 Δυαδική Αναζήτηση Παρεμβολής και η παραλλαγή της

Σε αυτήν την ενότητα θα εξετάσουμε τον αλγόριθμο αναζήτησης με παρεμβολή, καθώς επίσης και μια παραλλαγή του, η οποία αυξάνει την απόδοσή του.

3.3.1 Δυαδική Αναζήτηση Παρεμβολής

Θα ξεκινήσουμε αναλύοντας τον αλγόριθμο στην απλή του μορφή. Παρακάτω εξετάζουμε τις βασικές συναρτήσεις που χρησιμοποιούμε για την υλοποίησή του, καθώς επίσης και την ίδια τη συνάρτηση `bis`, η οποία υλοποιεί τον αλγόριθμο.

3.3.1.1 Η συνάρτηση `sqrt`

Δήλωση:

```
int sqrt(int num)
```

Παράμετροι:

- **num** – Ο αριθμός του οποίου την ρίζα θέλουμε να υπολογίσουμε.

Επιστρέφει:

Το ακέραιο μέρος από την ρίζα του **num**.

Η συνάρτηση **sqrt** χρησιμοποιείται για τον υπολογισμό της τετραγωνικής ρίζας του αριθμού **num**. Από τη ρίζα κρατάμε μόνο το ακέραιο κομμάτι. Ο υπολογισμός της ρίζας χρειάζεται στον αλγόριθμο `bis`, καθώς πρέπει να ελέγχουμε ομάδες στοιχείων μεγέθους \sqrt{size}

3.3.1.2 Η συνάρτηση `bis`

Δήλωση:

```
int* bis(Data* sortedData, Date date, int maxsize)
```

Παράμετροι:

- **sortedData** – Ο πίνακας στον οποίο θα γίνει η δυαδική αναζήτηση.
- **date** – Η ημερομηνία που αναζητάμε.
- **maxsize** – Ο μέγιστος αριθμός θέσης όλου του πίνακα.

Επιστρέφει:

Έναν πίνακα ακεραίων, όπου η πρώτη θέση περιέχει το μέγεθος του πίνακα και οι υπόλοιπες θέσεις περιέχουν τις θέσεις στο **sortedData** όλων των στοιχείων με την ημερομηνία **date**.

Η συνάρτηση **bis** εφαρμόζει τον αλγόριθμο της δυαδικής αναζήτησης παρεμβολής πάνω στα δεδομένα του ήδη ταξινομημένου πίνακα **sortedData**, αναζητώντας την ημερομηνία που καθορίζεται από το όρισμα **date**.

Η συνάρτηση αρχικά θέτει δύο μεταβλητές, τις **left** και **right**, στα αριστερά και δεξιά άκρα του **sortedData** αντίστοιχα.

Πριν αρχίσει η πρώτη εκτέλεση του αλγορίθμου, ελέγχουμε αν η ημερομηνία μπορεί να βρίσκεται εντός του πίνακα συγκρίνοντας την με το πρώτο και το τελευταίο στοιχείο, ώστε να αποφύγουμε Segmentation Fault. Αν η ημερομηνία **date** δεν βρίσκεται ανάμεσα στα αριστερά και δεξιά άκρα του πίνακα **data**, δεν μπορεί να περιέχεται στον πίνακα, οπότε η συνάρτηση επιστρέφει **NULL**.

Έπειτα υπολογίζεται η πρώτη τιμή του **next**, μεταβλητής του αλγορίθμου BIS.

Προτού ξεκινήσει η επαναληπτική διαδικασία, ελέγχουμε αν η τιμή του **next** βρίσκεται εντός των ορίων του πίνακα, καθώς αν η ημερομηνία που ψάχνουμε ταυτίζεται με την ημερομηνία του δεξιού άκρου του πίνακα το **next** θα πάρει την τιμή του **maxsize**, προκαλώντας Segmentation Fault, αφού ο πίνακας πάει μέχρι **maxsize - 1**. Αν συμβαίνει αυτό, στο **next** αναθέτουμε την τιμή του **right**, ώστε να είμαστε εντός του πίνακα.

Αρχίζει η επαναληπτική διαδικασία η οποία τρέχει έως ότου βρεθεί ένα στοιχείο με την ημερομηνία **date** ή το **right** γίνει μεγαλύτερο από το **left**, το οποίο σημαίνει ότι δεν υπάρχει στοιχείο με την ημερομηνία **date**.

Ξεκινώντας από το **next** η συνάρτηση ελέγχει ομάδες στοιχείων πλήθους $\sqrt{\text{size}}$ ώσπου βρει μια που μπορεί να περιέχει το στοιχείο που αναζητούμε. Όταν βρεθεί ενημερώνονται κατάλληλα τα **left**, **right** και **size**.

Έπειτα ελέγχουμε αν τα πεδία **date** των θέσεων **left** και **right** είναι ίσα, περίπτωση που, εξαιτίας του υπολογισμού του **next**, θα οδηγούσε σε διαίρεση με το μηδέν. Αν τα

πεδία **date** είναι ίσα, όποιο και να επιλέξουμε δεν υπάρχει διαφορά, οπότε επιλέγουμε το στοιχείο της θέσης **left** για να αποφύγουμε την διαίρεση με το μηδέν.

Τελικά, εφόσον δεν έχουμε διαίρεση με το μηδέν, υπολογίζουμε την επόμενη τιμή του **next**, και προχωρούμε στην επόμενη επανάληψη.

Όταν τελειώσει η επαναληπτική διαδικασία, ελέγχουμε αν βρέθηκε στοιχείο με την ημερομηνία **date**. Αν δεν βρέθηκε, επιστρέφουμε **NULL**.

Αλλιώς, αναζητούμε στα αριστερά και δεξιά του **next** ώστε να βρούμε όλα τα στοιχεία που έχουν την ημερομηνία **date**, κρατάμε τις θέσεις τους και τις επιστρέφουμε.

3.3.2 Παραλλαγή

Ο τροποποιημένος αλγόριθμος υλοποιείται με τη συνάρτηση **altbis**.

3.3.2.1 Η συνάρτηση **binpow**

Δήλωση:

```
int binpow(int num)
```

Παράμετροι:

- **num** – Η δύναμη στην οποία υψώνεται το δύο.

Επιστρέφει:

Το 2^{num} .

Η συνάρτηση **binpow** δέχεται ως όρισμα έναν αριθμό **num** και επιστρέφει το 2^{num} . Χρησιμοποιείται για τον υπολογισμό του εκθετικού βήματος που απαιτεί ο τροποποιημένος αλγόριθμος αναζήτησης με παρεμβολή.

3.3.2.2 Η συνάρτηση `altbis`

Δήλωση:

```
int* altbis(Data* sortedData, Date date, int maxsize)
```

Παράμετροι:

- **sortedData** – Ο πίνακας στον οποίο θα γίνει η δυαδική αναζήτηση.
- **date** – Η ημερομηνία που αναζητάμε.
- **maxsize** – Ο μέγιστος αριθμός θέσης όλου του πίνακα.

Επιστρέφει:

Έναν πίνακα ακεραίων, όπου η πρώτη θέση περιέχει το μέγεθος του πίνακα και οι υπόλοιπες θέσεις περιέχουν τις θέσεις στο **sortedData** όλων των στοιχείων με την ημερομηνία **date**.

Η συνάρτηση **altbis** υλοποιεί πάλι τον αλγόριθμο δυαδικής αναζήτησης παρεμβολής όπως τον περιγράψαμε, αλλά ελαφρώς τροποποιημένο. Συγκεκριμένα, για να βελτιώσουμε τον χρόνο χειρότερης περίπτωσης του BIS μπορούμε για τον υπολογισμό των νέων ορίων αναζήτησης αντί για το $i * \sqrt{\text{size}}$ να χρησιμοποιούμε το $(2^i) * \sqrt{\text{size}}$. Η τιμή 2^i υπολογίζεται με τη συνάρτηση **binpow**.

3.3.3 Πειραματικές συγκρίσεις και συμπεράσματα

Τώρα που είδαμε πώς λειτουργούν και οι δύο αλγόριθμοι αναζήτησης, ας συγκρίνουμε τα αποτελέσματά τους.

Αρχικά, γνωρίζουμε από τη θεωρία ότι στη χειρότερη περίπτωση ο αλγόριθμος δυαδικής αναζήτησης παρεμβολής εκτελείται σε χρόνο $O(\sqrt{n})$, ενώ η τροποποιημένη του μορφή βελτιώνεται σε $O(\log n)$. Περιμένουμε, επομένως, ο τροποποιημένος αλγόριθμος να είναι πιο αποδοτικός. Στη συνέχεια θα εκτελέσουμε τους δύο αλγορίθμους και θα εξάγουμε συμπεράσματα σχετικά με τον χρόνο εκτέλεσης. Για τον υπολογισμό του χρόνου εκτέλεσης, ακολουθούμε τη διαδικασία που περιγράψαμε στην Ενότητα 1.3.

Τρέχουμε τους δύο αλγορίθμους για την ημερομηνία 1/2/2002:

```
Day:
Month:
Year:
BIS - Temperature: 14.730
Phosphate: 0.470

BIS Search Time: 0.000003

Alt BIS - Temperature: 14.730
Phosphate: 0.470

Alt BIS Search Time: 0.000001
```

Εικόνα 3.6 Εκτέλεση των αλγορίθμων για την ημερομηνία 1/2/2002

Παρατηρούμε ότι ο αλγόριθμος με το γραμμικό βήμα απαιτεί χρόνο 3 μ s, ενώ ο αλγόριθμος με το εκθετικό βήμα απαιτεί μόνο 1 μ s. Επιβεβαιώνουμε ότι είναι πράγματι γρηγορότερος.

Κεφάλαιο 4

Δέντρα AVL

Στο κεφάλαιο αυτό υλοποιούμε κάποιους βασικούς αλγόριθμους για τη διαχείριση AVL δέντρων, με σκοπό να μπορούμε να φορτώσουμε και να επεξεργαστούμε το αρχείο **ocean.csv** σε ένα δέντρο AVL. Αρχικά θα εξετάσουμε τον τρόπο με τον οποίο αποθηκεύουμε τα δεδομένα στο πρόγραμμά μας και στη συνέχεια θα δούμε λεπτομερώς συναρτήσεις που υλοποιούν πράξεις όπως η αναζήτηση, η ένθεση και η διαγραφή, μαζί με τις διάφορες διορθωτικές πράξεις που αυτές απαιτούν, καθώς επίσης και διάφορες διαπεράσεις δέντρων.

Ενότητα 4.1 Αναπαράσταση των δεδομένων

Όπως είδαμε στο Κεφάλαιο 1, έτσι και εδώ χρειαζόμαστε έναν τρόπο να αποθηκεύουμε τα δεδομένα του αρχείου **ocean.csv**, αυτή τη φορά σε ένα δέντρο AVL. Υπενθυμίζουμε ότι το αρχείο **ocean.csv** αποτελείται από γραμμές, όπου κάθε γραμμή-δείγμα αντιστοιχεί σε μια μέτρηση της θερμοκρασίας (σε °C) στο νερό του ωκεανού καθώς και των τιμών Phosphate, Silicate, Nitrite, Nitrate, Salinity, Oxygen για την περίοδο 2000 έως 2019. Η δομή κάθε γραμμής είναι:

Date	Temp	Phosphate	Silicate	Nitrite	Nitrate	Salinity	Oxygen
------	------	-----------	----------	---------	---------	----------	--------

Εικόνα 4.1 Δομή μιας γραμμής του ocean.csv

Προφανώς θα θεωρήσουμε ότι κάθε γραμμή του αρχείου αντιστοιχεί σε κάποιον κόμβο του δέντρου. Το AVL δέντρο που θα υλοποιήσουμε είναι φυλλοπροσανατολισμένο, δηλαδή κάθε γραμμή αποθηκεύεται σε ένα από τα φύλλα, ενώ οι εσωτερικοί κόμβοι περιέχουν βοηθητική πληροφορία.

Θα χρειαστεί να υλοποιήσουμε δύο είδη AVL δέντρων στο πρόγραμμά μας, ένα που είναι ταξινομημένο ως προς τις ημερομηνίες και ένα που είναι ταξινομημένο ως προς τις θερμοκρασίες. Στη συνέχεια αυτά τα δύο είδη AVL θα αποκαλούνται AVL ημερομηνιών και AVL θερμοκρασιών αντίστοιχα για λόγους ευκολίας. Και τα δύο αξιοποιούν την ίδια δομή για τους κόμβους τους. Θα δούμε λεπτομέρειες σχετικά με τη δομή των κόμβων παρακάτω.

Σε περίπτωση που δύο ή περισσότερες γραμμές του αρχείου **ocean.csv** έχουν την ίδια ημερομηνία (για AVL ημερομηνιών) ή την ίδια θερμοκρασία (για AVL θερμοκρασιών), δεν τις αγνοούμε, αλλά εισάγουμε την παραπάνω πληροφορία στο ήδη υπάρχον φύλλο. Για παράδειγμα, αν δύο γραμμές έχουν την ίδια ημερομηνία **d**, αλλά διαφορετικές (ή ακόμα και ίδιες) θερμοκρασίες **t1** και **t2**, τότε οφείλουμε να έχουμε στο AVL ημερομηνιών ένα φύλλο με την ημερομηνία **d** και τις δύο θερμοκρασίες **t1**, **t2** σε έναν πίνακα 2 θέσεων. Η περίπτωση για τα AVL θερμοκρασιών είναι αντίστοιχη.

Τώρα που αναδείξαμε τις επιλογές υλοποίησής μας, θα εξετάσουμε τη δομή ενός AVL κόμβου. Για την αναπαράσταση των κόμβων δημιουργούμε ένα δικό μας struct με όνομα **AVLnode**, το οποίο διαθέτει τα παρακάτω πεδία:

```
struct AVLnode
{
    Date* date;
    double* temperature;
    int mult;
    struct AVLnode *father;
    struct AVLnode *leftChild;
    struct AVLnode *rightChild;
    int height;
    int hb;
};
```

Έστω ένας τυχαίος κόμβος του AVL δέντρου, δηλαδή ένα στιγμιότυπο της παραπάνω δομής. Τότε:

- Το πεδίο **date** είναι ένας δείκτης προς την ημερομηνία (struct τύπου **Date**) που είναι αποθηκευμένη στον συγκεκριμένο κόμβο. Θεωρούμε ότι είναι ένας δυναμικά δεσμευμένος πίνακας ημερομηνιών, ο οποίος στην περίπτωση του AVL ημερομηνιών περιέχει μόνο μια ημερομηνία, ενώ στην περίπτωση του AVL θερμοκρασιών μπορεί να περιέχει περισσότερες ημερομηνίες, οι οποίες θα έχουν όλες την ίδια θερμοκρασία.
- Το πεδίο **temperature** είναι ένας δείκτης προς την θερμοκρασία που είναι αποθηκευμένη στον συγκεκριμένο κόμβο. Θεωρούμε ότι είναι ένας δυναμικά δεσμευμένος πίνακας πραγματικών, ο οποίος στην περίπτωση του AVL θερμοκρασιών περιέχει μόνο μια τιμή, ενώ στην περίπτωση του AVL ημερομηνιών μπορεί να περιέχει περισσότερες θερμοκρασίες, οι οποίες θα έχουν όλες την ίδια ημερομηνία.
- Το πεδίο **mult** (multiplicity) δείχνει πόσες φορές εισάχθηκε η ίδια ημερομηνία στο AVL ημερομηνιών ή η ίδια θερμοκρασία στο AVL θερμοκρασιών. Όπως εξηγήσαμε, θα πρέπει κάθε φορά που εισάγουμε μια ήδη υπάρχουσα τιμή στο AVL, έστω μια ίδια ημερομηνία, να αποθηκεύουμε στον υπάρχοντα κόμβο όλες τις θερμοκρασίες που αντιστοιχούν σ' αυτήν την ημερομηνία (αντίστοιχα αν εισάγουμε θερμοκρασίες). Επομένως, στην περίπτωση του AVL ημερομηνιών, το

πεδίο αυτό μάς δείχνει το μέγεθος του πίνακα **temperature**, ενώ στην περίπτωση του AVL θερμοκρασιών μάς δείχνει το μέγεθος του πίνακα **date**.

- Τα πεδία **father**, **leftChild** και **rightChild** είναι δείκτες προς τον πατέρα, το αριστερό παιδί και το δεξιό παιδί του κόμβου στο AVL δέντρο αντίστοιχα.
- Το πεδίο **height** είναι το ύψος (≥ 0) του υποδέντρου με ρίζα τον συγκεκριμένο κόμβο. Για κάθε φύλλο ισχύει **height** = 0, ενώ για κάθε εσωτερικό κόμβο πρέπει να ισχύει **height** = $\max\{\text{leftChild} \rightarrow \text{height}, \text{rightChild} \rightarrow \text{height}\} + 1$
- Το πεδίο **hb** (height balance) αποτελεί την υψοζύγηση του συγκεκριμένου κόμβου και υπολογίζεται σε κάθε περίπτωση ως **hb** = **rightChild** \rightarrow **height** - **leftChild** \rightarrow **height**. Σύμφωνα με αυτό το πεδίο αποφασίζουμε πότε πρέπει να πραγματοποιηθεί μια διορθωτική πράξη στο AVL δέντρο ύστερα από μια ένθεση ή διαγραφή.

Τώρα που είδαμε πώς υλοποιείται ένας κόμβος του AVL, το δέντρο μας δε θα είναι τίποτα άλλο παρά ένας δείκτης προς τη ρίζα, δηλαδή ένας δείκτης τύπου **AVLnode**:

```
AVLnode *tree = NULL;
```

Όπως βλέπουμε, αυτός ο δείκτης έχει αρχικά την τιμή **NULL** επειδή το δέντρο είναι κενό. Παρακάτω θα δούμε πώς δημιουργούμε ένα δέντρο από τα δεδομένα του **ocean.csv**, ώστε έπειτα να δώσουμε στη μεταβλητή **tree** τη διεύθυνση της ρίζας. Αν στο τέλος του προγράμματος ή μετά από επανειλημμένες διαγραφές το δέντρο διαγραφεί πλήρως, τότε η τιμή αυτού του δείκτη οφείλει να γίνει πάλι **NULL**.

Ενότητα 4.2 Αποθήκευση των δεδομένων

Αφού είδαμε πώς μπορούμε να αναπαραστήσουμε τα δεδομένα μας μέσα στο πρόγραμμα, ας δούμε τώρα πώς γίνεται η μετατροπή των δεδομένων από τη μορφή που περιέχονται στο αρχείο στη μορφή που επιθυμούμε.

Η μετατροπή των δεδομένων γίνεται με παρόμοιο τρόπο με αυτόν που παρουσιάσαμε στο Κεφάλαιο 1. Όπως και πριν, θα χρειαστεί να διαβάζουμε κάθε γραμμή του **ocean.csv** και να εξάγουμε την πληροφορία που χρειαζόμαστε. Σε αντίθεση με το Μέρος Ι, εδώ χρειαζόμαστε μόνο την ημερομηνία και τη θερμοκρασία από κάθε γραμμή, δηλαδή τις δύο πρώτες τιμές. Για τον σκοπό αυτό χρησιμοποιούμε τη συνάρτηση **readDateandTemp**, η οποία είναι μια τροποποιημένη εκδοχή της **readCsv** και αναλύεται στη συνέχεια.

4.2.1 Η συνάρτηση readDateAndTemp

Δήλωση:

```
void readDateAndTemp(char *filename, AVLnode **tree, int treeType)
```

Παράμετροι:

- **filename** – Το όνομα του .csv αρχείου από το οποίο θα κατασκευαστεί το AVL δέντρο.
- **tree** – Διεύθυνση του κενού δέντρου στο οποίο θα τοποθετηθεί το νέο AVL.
- **treeType** – Καθορίζει το είδος του δέντρου που θα δημιουργηθεί.

Η συνάρτηση **readDateAndTemp** ανοίγει και διαβάζει το .csv αρχείο που δίνεται από το όρισμα **filename** και δημιουργεί ένα AVL δέντρο που στα φύλλα του περιέχει τα δεδομένα του αρχείου. Είναι αντίστοιχη της συνάρτησης **readCsv**, αλλά για AVL δέντρα. Το είδος του δέντρου που θα δημιουργηθεί καθορίζεται από το όρισμα **treeType**:

- Για **treeType** = 0, δημιουργείται ένα AVL ημερομηνιών.
- Για **treeType** ≠ 0, δημιουργείται ένα AVL θερμοκρασιών.

Για να αρχίσει επαναληπτικά η ανάγνωση του αρχείου, πρέπει πρώτα να καλέσουμε μια φορά τη συνάρτηση **readLine** χωρίς να κρατάμε κάπου το αποτέλεσμα, ώστε να μετακινηθεί ο δείκτης του αρχείου από την πρώτη γραμμή που περιέχει άχρηστη πληροφορία στην αρχή της επόμενης γραμμής. Δεν ξεχνάμε φυσικά να αποδεσμεύσουμε τη μνήμη που δεσμεύσαμε για την άχρηστη γραμμή που διαβάσαμε:

```
//Get rid of the first line before reading the other ones
free(readLine(newFile, linesize));
```

Έπειτα μπορούμε με μια επαναληπτική διαδικασία να διαβάζουμε γραμμές από το αρχείο:

```
while (i < size)
{
    char *line = readLine(file, linesize);

    .
    .
    .
}
```

Από κάθε γραμμή που διαβάζεται, κρατάμε μόνο τις δύο πρώτες τιμές, την ημερομηνία και τη θερμοκρασία. Παρατηρούμε ότι αυτές διαχωρίζονται με κόμμα (,), άρα καλούμε την **strtok** με διαχωριστή το κόμμα και κρατώντας έναν μετρητή σταματάμε μόλις διαβαστούν οι δύο τιμές που θέλουμε. Την ημερομηνία την αποθηκεύουμε σε ένα

ξεχωριστό αλφαριθμητικό ώστε να τη διαχειριστούμε στη συνέχεια, ενώ τη θερμοκρασία τη μετατρέπουμε από αλφαριθμητικό σε πραγματικό με τη συνάρτηση **atof**:

```
char *token = strtok(line, ","); // it separates the array and the
separator is the ',' character

while (counter < 2)
{
    if (counter == 0)
    {
        dateString = (char *)malloc((strlen(token) + 1) *
sizeof(char)); // given the length of the line tha we raed ,we
allocate memory for the string object
        strcpy(dateString, token);
        // and then we copy it
    }
    else
        temp = atof(token); // save the other data in the array

    token = strtok(NULL, ","); // line of code that exists for strtok
to work properly :) (basically works in the same string )
    counter++; // counter for the data of the array
}
i++;
```

Στο τέλος διαχωρίζουμε ξεχωριστά το αλφαριθμητικό **dateString**, το οποίο χρησιμοποιεί ως διαχωριστή την κάθετο (/). Τα επιμέρους κομμάτια αποθηκεύονται σε ένα struct τύπου **Date**:

```
counter = 0;

token = strtok(dateString, "/"); // same procidure like above but now
we want ot store the values of the date
while (token)
{
    if (counter == 0) // for these lines we make the values double
and save them in the wright spot of the array
        date.month = atoi(token);
    else if (counter == 1)
        date.day = atoi(token);
    else
        date.year = atoi(token);

    token = strtok(NULL, "/");
    counter++;
}
```

Αφού η συνάρτηση εξάγει αυτές τις δύο τιμές από μια συγκεκριμένη γραμμή, καλεί μια από τις συνάρτησεις **insertDateAVLnode** ή **insertTempAVLnode**, ανάλογα με τον αν το AVL δέντρο είναι διατεταγμένο ως προς τις ημερομηνίες ή τις θερμοκρασίες αντίστοιχα (οι συναρτήσεις αυτές αναλύονται σε επόμενη ενότητα):

```
if (treeType == 0)
    insertDateAVLnode(tree, temp, date);
else insertTempAVLnode(tree, temp, date);
```

Αφού διαβαστούν όλες οι γραμμές του αρχείου, το AVL που δημιουργήθηκε, δηλαδή η ρίζα του, αποθηκεύεται τελικά στη μεταβλητή του ορίσματος **tree**.

Για να λειτουργήσει σωστά η συνάρτηση, το .csv αρχείο από το οποίο γίνεται η ανάγνωση πρέπει να βρίσκεται στην αρχική του μορφή, δηλαδή να περιέχει μια γραμμή με άχρηστη πληροφορία στην αρχή και ακριβώς μια κενή γραμμή στο τέλος:

- Η έλλειψη της πρώτης, άχρηστης γραμμής του αρχείου οδηγεί στην αγνόηση της πρώτης μέτρησης, καθώς πάντα ξεφορτωνόμαστε την πρώτη γραμμή του αρχείου.
- Η έλλειψη της τελευταίας, κενής γραμμής του αρχείου οδηγεί στην αγνόηση της τελευταίας μέτρησης.
- Η ύπαρξη παραπάνω κενών γραμμών στο τέλος του αρχείου οδηγεί σε σφάλμα.

4.2.2 Η συνάρτηση freeTree

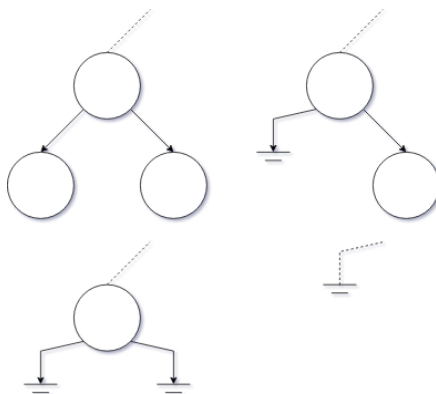
Δήλωση:

```
void freeTree (AVLnode **tree)
```

Παράμετροι:

- **tree** - Δείκτης προς το δέντρο που πρόκειται να διαγραφεί από τη μνήμη.

Η συνάρτηση **freeTree** διαγράφει τους κόμβους του AVL δέντρου από τη μνήμη, οι οποίοι είχαν προηγουμένως δημιουργηθεί με δυναμική δέσμευση μνήμης, ξεκινώντας από τα φύλλα και φτάνοντας τελικά μέχρι την ρίζα. Αυτό επιτυγχάνεται εφαρμόζοντας αναδρομικά μεταδιατεταγμένη διαπέραση πάνω στο δέντρο, ώστε να εγγυόμαστε ότι πρώτα θα απελευθερώνονται τα φύλλα στα οποία έχουμε πρόσβαση και τελικά ο πατέρας τους.



Εικόνα 4.2 Διαγραφή υποδέντρου με μεταδιατεταγμένη διαπέραση

Πριν γίνει η διαγραφή της ρίζας του υποδέντρου απελευθερώνονται και τα πεδία **date** και **temperature**, τα οποία είναι δυναμικά δεσμευμένοι πίνακες. Μετά τη διαγραφή της ρίζας, η τιμή του δείκτη προς αυτήν αλλάζει σε **NULL**. Αυτό έχει ως αποτέλεσμα:

- Αν ο κόμβος αποτελεί παιδί ενός άλλου κόμβου, ο δείκτης προς αυτό το παιδί να γίνεται **NULL**.
- Αν ο κόμβος είναι η ρίζα ολόκληρου του δέντρου, το δέντρο να αποκτά την τιμή **NULL**.

Ενότητα 4.3 Περιστροφές

Πριν δούμε πώς υλοποιούνται κάποιες βασικές πράξεις πάνω στα δέντρα, όπως η ένθεση και η διαγραφή, θα δούμε πρώτα τις διορθωτικές πράξεις που αυτές χρειάζονται. Στο πρόγραμμά μας έχουμε υλοποιήσει 4 είδη περιστροφών, 2 απλές και 2 διπλές.

4.3.1 Απλή περιστροφή

Υπάρχουν δύο είδη απλών περιστροφών, η αριστερή περιστροφή και η δεξιά περιστροφή. Η πρώτη υλοποιείται με τη συνάρτηση **leftRotation**, ενώ η δεύτερη με τη συνάρτηση **rightRotation**.

4.3.1.1 Η συνάρτηση `leftRotation`

Δήλωση:

```
AVLnode *leftRotation (AVLnode *node)
```

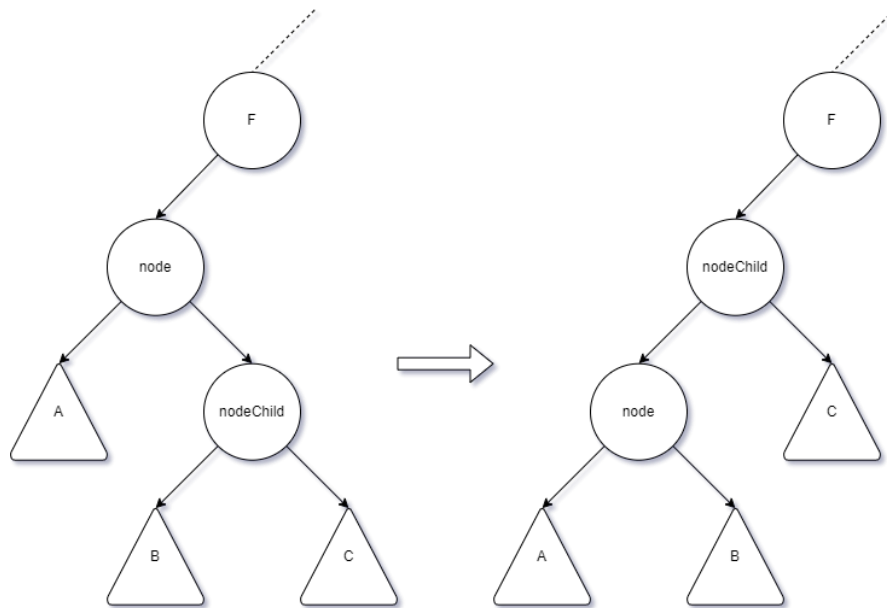
Παράμετροι:

- **node** – Ο κόμβος γύρω από τον οποίο γίνεται η περιστροφή.

Επιστρέφει:

Τη νέα ρίζα του υποδέντρου με αρχική ρίζα το **node** μετά την περιστροφή του **node**.

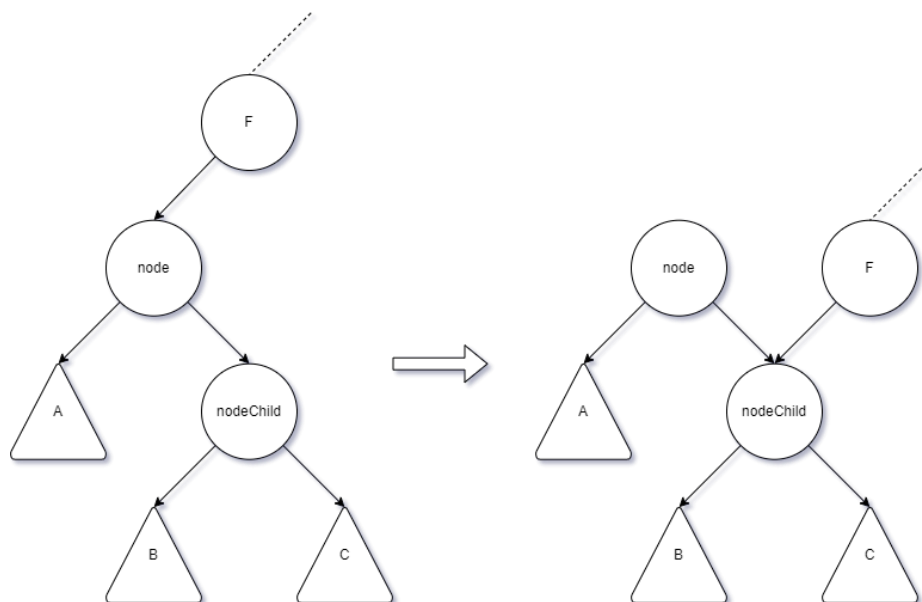
Η συνάρτηση **leftRotation** εκτελεί μια αριστερή περιστροφή γύρω από τον κόμβο **node**. Μετά την αριστερή περιστροφή, θα πρέπει το δεξιό παιδί του κόμβου **node**, το οποίο αποκαλούμε **nodeChild**, να γίνει η νέα ρίζα του υποδέντρου, ενώ ο κόμβος **node** να γίνει αριστερό παιδί του **nodeChild** και να αποκτήσει ως νέο δεξιό παιδί το αρχικό αριστερό παιδί του **nodeChild**. Η διαδικασία που περιγράψαμε φαίνεται στο παρακάτω σχήμα:



Εικόνα 4.3 Αριστερή περιστροφή γύρω από τον κόμβο node

Έστω ότι θέλουμε να κάνουμε περιστροφή το αριστερό δέντρο της παραπάνω εικόνας. Η διαδικασία της περιστροφής εκτελείται στη συνάρτηση 3 βήματα:

1. Αν ο **node** έχει μη κενό πατέρα **F**, πρέπει να θέσουμε ως νέο παιδί του **F** τον κόμβο **nodeChild**:



Εικόνα 4.4 Βήμα 1 της αριστερής περιστροφής

```
if (node->father != NULL)
{
    nodeFather = node->father;
```

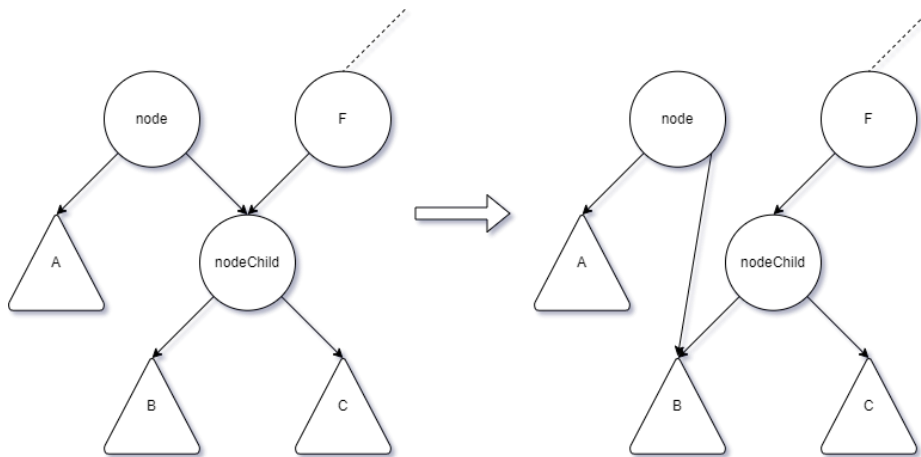
```

if (node == nodeFather->leftChild)
    nodeFather->leftChild = nodeChild;
else if (node == nodeFather->rightChild)
    nodeFather->rightChild = nodeChild;

nodeChild->father = nodeFather;
}

```

2. Θέτουμε ως δεξιό παιδί του **node** το αριστερό παιδί του **nodeChild**.



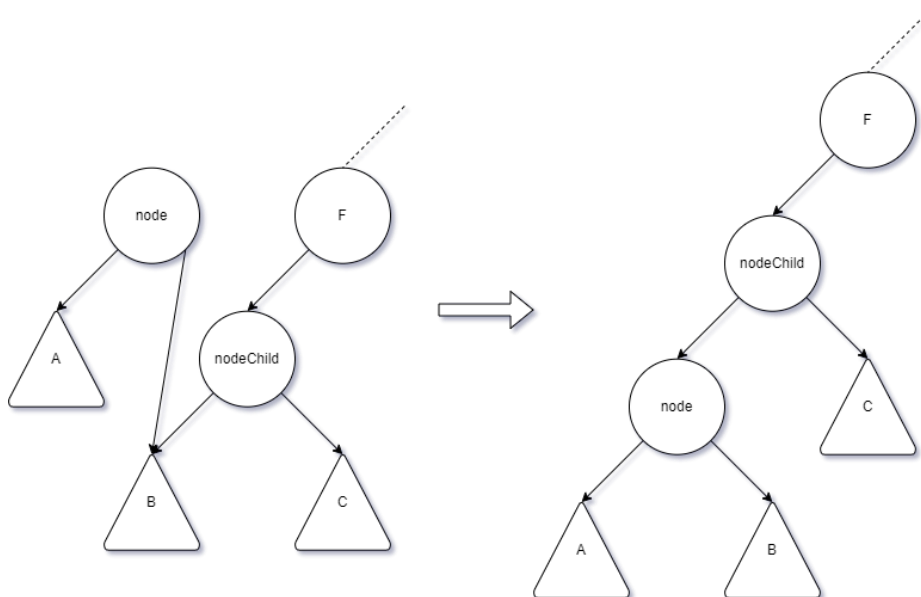
Εικόνα 4.5 Βήμα 2 της αριστερής περιστροφής

```

node->rightChild = nodeChild->leftChild;
nodeChild->leftChild->father = node;

```

3. Θέτουμε ως αριστερό παιδί του **nodeChild** το **node**.



Εικόνα 4.6 Βήμα 3 της αριστερής περιστροφής

```
nodeChild->leftChild = node;
node->father = nodeChild;
```

Αφού γίνει η αλλαγή των δεικτών, πρέπει να υπολογίσουμε τα νέα ύψη και τις υψοζυγίσεις των κόμβων `node` και `nodeChild`, ξεκινώντας από τον χαμηλότερο κόμβο (που είναι ο `node`). Υπενθυμίζουμε ότι το ύψος υπολογίζεται ως `max{leftChild->height, rightChild->height} + 1`, ενώ η υψοζύγιση `rightChild->height - leftChild->height`.

Στο τέλος της συνάρτησης, επιστρέφεται η νέα ρίζα του υποδέντρου, δηλαδή ο κόμβος `nodeChild`, ούτως ώστε αν αυτός αποτελεί τη ρίζα όλου του δέντρου, να κάνουμε τον δείκτη προς τη ρίζα να δείχνει προς αυτόν:

```
return nodeChild;
```

4.3.1.2 Η συνάρτηση `rightRotation`

Δήλωση:

```
AVLnode *rightRotation(AVLnode *node)
```

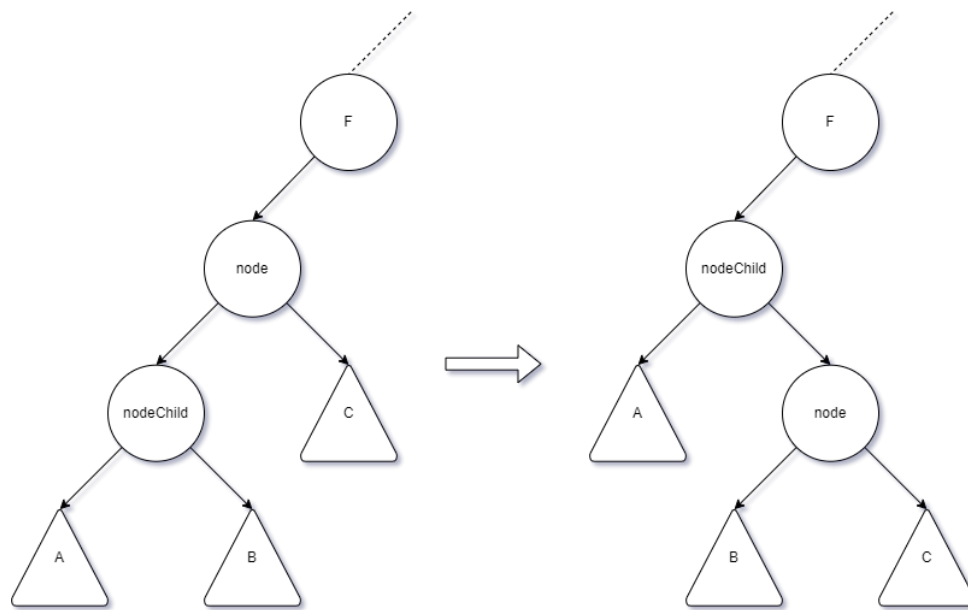
Παράμετροι:

- **node** – Ο κόμβος γύρω από τον οποίο γίνεται η περιστροφή.

Επιστρέφει:

Τη νέα ρίζα του υποδέντρου με αρχική ρίζα το **node** μετά την περιστροφή του **node**.

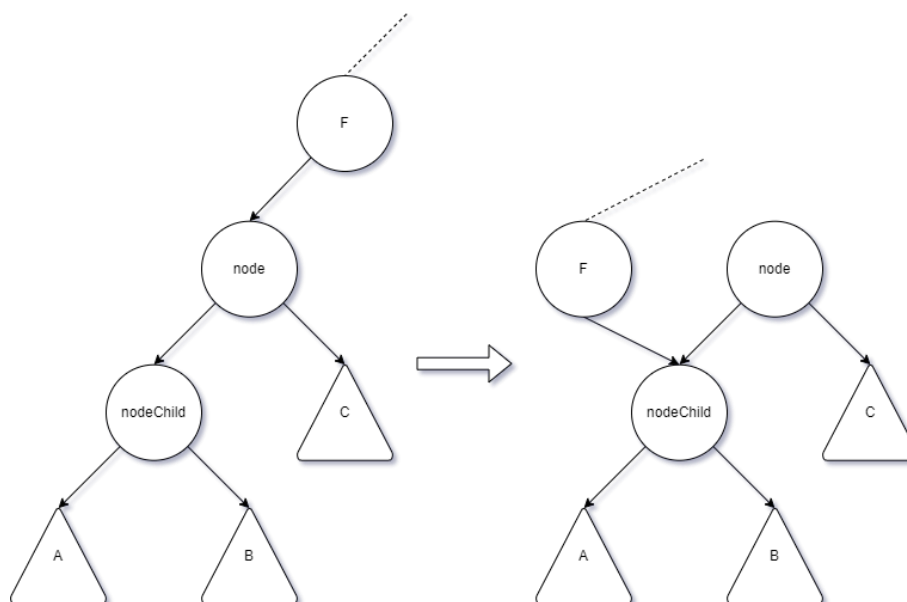
Η συνάρτηση **rightRotation** εκτελεί μια δεξιά περιστροφή γύρω από τον κόμβο `node`. Μετά την δεξιά περιστροφή, θα πρέπει το αριστερό παιδί του κόμβου `node`, το οποίο αποκαλούμε `nodeChild`, να γίνει η νέα ρίζα του υποδέντρου, ενώ ο κόμβος `node` να γίνει δεξιό παιδί του `nodeChild` και να αποκτήσει ως νέο αριστερό παιδί το αρχικό δεξιό παιδί του `nodeChild`. Η διαδικασία που περιγράψαμε φαίνεται στο παρακάτω σχήμα:



Εικόνα 4.7 Δεξιά περιστροφή γύρω από τον κόμβο node

Έστω ότι θέλουμε να κάνουμε περιστροφή το αριστερό δέντρο της παραπάνω εικόνας. Η διαδικασία της περιστροφής εκτελείται στη συνάρτηση 3 βήματα:

1. Αν ο **node** έχει μη κενό πατέρα **F**, πρέπει να θέσουμε ως νέο παιδί του **F** τον κόμβο **nodeChild**:



Εικόνα 4.8 Βήμα 1 της δεξιάς περιστροφής

```
if (node->father != NULL)
{
    nodeFather = node->father;
```

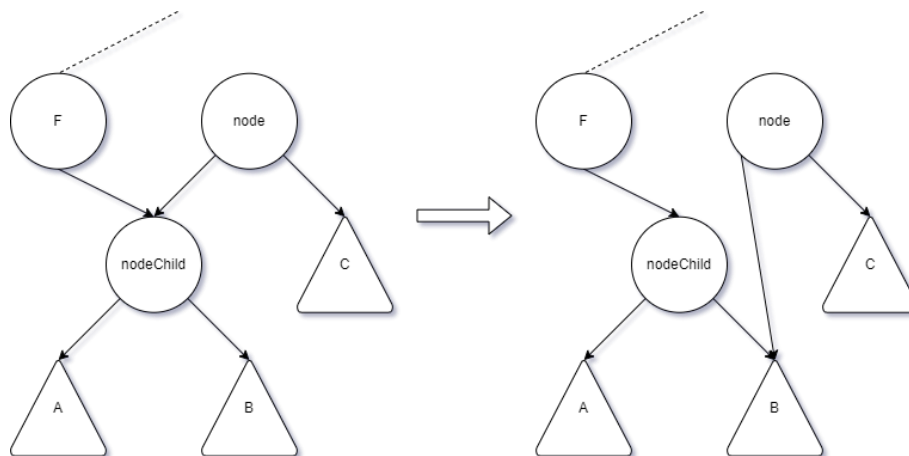
```

if (node == nodeFather->leftChild)
    nodeFather->leftChild = nodeChild;
else if (node == nodeFather->rightChild)
    nodeFather->rightChild = nodeChild;

nodeChild->father = nodeFather;
}

```

2. Θέτουμε ως αριστερό παιδί του **node** το δεξιό παιδί του **nodeChild**.



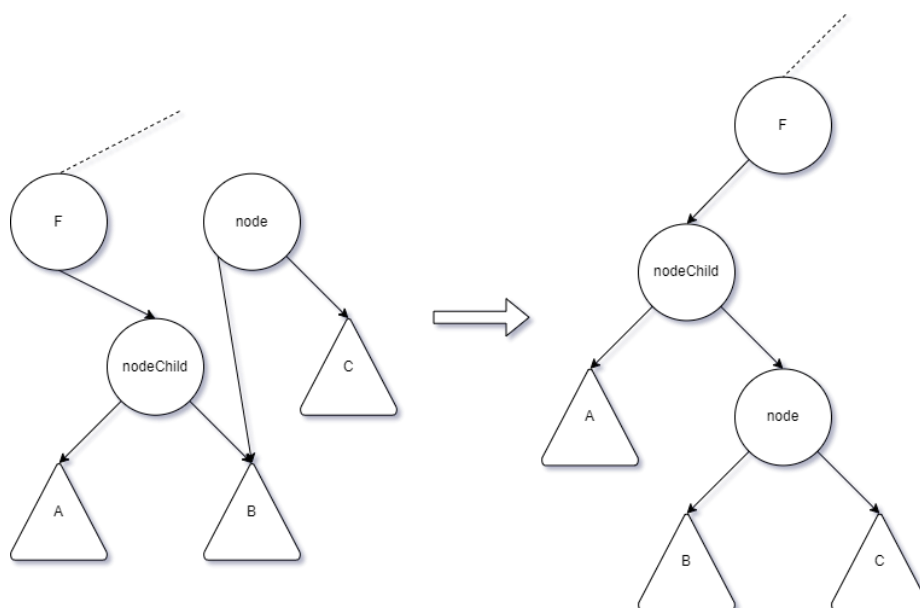
Εικόνα 4.9 Βήμα 2 της δεξιάς περιστροφής

```

node->leftChild = nodeChild->rightChild;
nodeChild->rightChild->father = node;

```

3. Θέτουμε ως δεξιό παιδί του **nodeChild** το **node**.



Εικόνα 4.10 Βήμα 3 της δεξιάς περιστροφής

```
nodeChild->rightChild = node;  
node->father = nodeChild;
```

Αφού γίνει η αλλαγή των δεικτών, πρέπει να υπολογίσουμε τα νέα ύψη και τις υψοζυγίσεις των κόμβων `node` και `nodeChild`, ξεκινώντας από τον χαμηλότερο κόμβο (που είναι ο `node`). Υπενθυμίζουμε ότι το ύψος υπολογίζεται ως `max{leftChild->height, rightChild->height} + 1`, ενώ η υψοζύγιση `rightChild->height - leftChild->height`.

Στο τέλος της συνάρτησης, επιστρέφεται η νέα ρίζα του υποδέντρου, δηλαδή ο κόμβος `nodeChild`, ούτως ώστε αν αυτός αποτελεί τη ρίζα όλου του δέντρου, να κάνουμε τον δείκτη προς τη ρίζα να δείχνει προς αυτόν:

```
return nodeChild;
```

4.3.2 Διπλή περιστροφή

Υπάρχουν δύο είδη διπλών περιστροφών, η αριστερή-δεξιά περιστροφή και η δεξιά-αριστερή περιστροφή. Η πρώτη υλοποιείται με τη συνάρτηση `leftRightRotation`, ενώ η δεύτερη με τη συνάρτηση `rightLeftRotation`.

4.3.2.1 Η συνάρτηση `leftRightRotation`

Δήλωση:

```
AVLnode *leftRightRotation(AVLnode *node)
```

Παράμετροι:

- **node** – Ο κόμβος γύρω από τον οποίο γίνεται η περιστροφή.

Επιστρέφει:

Τη νέα ρίζα του υποδέντρου με αρχική ρίζα το **node** μετά την περιστροφή του **node**.

Η συνάρτηση `leftRightRotation` εκτελεί μια αριστερή-δεξιά περιστροφή γύρω από τον κόμβο **node**. Αποτελεί σύνθεση των συναρτήσεων `leftRotation` και `rightRotation`, καθώς υλοποιείται με μια εκτέλεση της `rightRotation` για το δεξιό παιδί του **node**, ακολουθούμενη από μια εκτέλεση της `leftRotation` για τον ίδιο τον **node**:

```
AVLnode *nodeChild = node->rightChild;

rightRotation(nodeChild);
return leftRotation(node);
```

Η νέα ρίζα του υποδέντρου, δηλαδή το αποτέλεσμα της **leftRotation**, επιστρέφεται από τη συνάρτηση, ώστε να πράξουμε κατάλληλα σε περίπτωση που είναι η ρίζα και ολόκληρου του δέντρου.

4.3.2.2 Η συνάρτηση `rightLeftRotation`

Δήλωση:

```
AVLnode *rightLeftRotation(AVLnode *node)
```

Παράμετροι:

- **node** – Ο κόμβος γύρω από τον οποίο γίνεται η περιστροφή.

Επιστρέφει:

Τη νέα ρίζα του υποδέντρου με αρχική ρίζα το **node** μετά την περιστροφή του **node**.

Η συνάρτηση **rightLeftRotation** εκτελεί μια δεξιά-αριστερή περιστροφή γύρω από τον κόμβο **node**. Αποτελεί σύνθεση των συναρτήσεων **leftRotation** και **rightRotation**, καθώς υλοποιείται με μια εκτέλεση της **leftRotation** για το αριστερό παιδί του **node**, ακολουθούμενη από μια εκτέλεση της **rightRotation** για τον ίδιο τον **node**:

```
AVLnode *nodeChild = node->leftChild;

leftRotation(nodeChild);
return rightRotation(node);
```

Η νέα ρίζα του υποδέντρου, δηλαδή το αποτέλεσμα της **rightRotation**, επιστρέφεται από τη συνάρτηση, ώστε να πράξουμε κατάλληλα σε περίπτωση που είναι η ρίζα και ολόκληρου του δέντρου.

Ενότητα 4.4 Βασικές πράξεις πάνω σε δέντρα AVL

Τώρα που είδαμε πώς υλοποιούνται οι περιστροφές στο πρόγραμμά μας, θα εξετάσουμε τις πράξεις της εύρεσης, της ένθεσης και της διαγραφής στα δέντρα AVL.

4.4.1 Αναζήτηση

Πρώτα θα δούμε πώς υλοποιείται η αναζήτηση ενός κόμβου σε ένα δέντρο AVL, καθώς η ένθεση και η διαγραφή απαιτούν πρώτα μια αναζήτηση για να βρεθεί ο κόμβος όπου θα γίνει ένθεση ή ο κόμβος που θα διαγραφεί. Υλοποιούμε δύο είδη αναζήτησης, μια για AVL ημερομηνιών που γίνεται με τη συνάρτηση **dateSearch** και μια για AVL θερμοκρασιών που γίνεται με τη συνάρτηση **tempSearch**. Και οι δύο αυτές συναρτήσεις αναλύονται στη συνέχεια.

4.4.1.1 Η συνάρτηση dateSearch

Δήλωση:

```
AVLnode *dateSearch (AVLnode *tree, Date date, int *found)
```

Παράμετροι:

- **tree** – Το AVL ημερομηνιών μέσα στο οποίο θα κάνουμε την αναζήτηση.
- **date** – Την ημερομηνία που αναζητάμε.
- **found** – Ακέραιος που δηλώνει αν το στοιχείο βρέθηκε ή όχι μέσα στο δέντρο.

Επιστρέφει:

Το φύλλο στο οποίο καταλήγει η αναζήτηση.

Η συνάρτηση **dateSearch** αναζητά στο δέντρο ημερομηνιών **tree** ένα φύλλο με κλειδί την ημερομηνία **date**. Εφαρμόζει τη γνώστη μέθοδο αναζήτησης σε φυλλοπροσανατολισμένα δυαδικά δέντρα, όπου αν η ζητούμενη τιμή είναι μικρότερη ή ίση από το κλειδί ενός εξεταζόμενου κόμβου, τότε μετακινούμαστε προς το αριστερό του υποδέντρο, ενώ αν είναι μεγαλύτερη, μετακινούμαστε προς το δεξιό του υποδέντρο. Η αναζήτηση λήγει όταν φτάσουμε σε ένα φύλλο, έστω **leaf**, όπου εξετάζουμε δύο ενδεχόμενα:

- Αν **date = leaf.date**, τότε **found = 1**
- Αν **date ≠ leaf.date**, τότε **found = 0**

Το όρισμα **found** περνάει στη συνάρτηση με αναφορά, ώστε η υπολογιζόμενη τιμή του να διατηρείται και έξω από τη συνάρτηση. Δεν μπορούμε απλά να επιστρέψουμε την τιμή του, καθώς χρειάζεται στο τέλος να επιστρέψουμε το φύλλο **leaf** στο οποίο έληξε η αναζήτηση. Αυτό είναι απαραίτητο, καθώς θα πρέπει να ξέρουμε σε ποιον κόμβο θα εφαρμοστούν οι πράξεις της ένθεσης και της διαγραφής, οι οποίες απαιτούν πρώτα ένα βήμα αναζήτησης στο δέντρο.

4.4.1.2 Η συνάρτηση tempSearch

Δήλωση:

```
AVLnode *tempSearch(AVLnode *tree, double temp, int *found)
```

Παράμετροι:

- **tree** – Το AVL θερμοκρασιών μέσα στο οποίο θα κάνουμε την αναζήτηση.
- **temp** – Η θερμοκρασία που αναζητάμε.
- **found** – Ακέραιος που δηλώνει αν το στοιχείο βρέθηκε ή όχι μέσα στο δέντρο.

Επιστρέφει:

Το φύλλο στο οποίο καταλήγει η αναζήτηση.

Η συνάρτηση **tempSearch** αναζητά στο δέντρο θερμοκρασιών *tree* ένα φύλλο με κλειδί τη θερμοκρασία **temp**. Εφαρμόζει τη γνώστη μέθοδο αναζήτησης σε φυλλοπροσανατολισμένα δυαδικά δέντρα, όπου αν η ζητούμενη τιμή είναι μικρότερη ή ίση από το κλειδί ενός εξεταζόμενου κόμβου, τότε μετακινούμαστε προς το αριστερό του υποδέντρο, ενώ αν είναι μεγαλύτερη, μετακινούμαστε προς το δεξιό του υποδέντρο. Η αναζήτηση λήγει όταν φτάσουμε σε ένα φύλλο, έστω **leaf**, όπου εξετάζουμε δύο ενδεχόμενα:

- Αν **temp = leaf.temp**, τότε **found = 1**
- Αν **temp ≠ leaf.temp**, τότε **found = 0**

Το όρισμα **found** περνάει στη συνάρτηση με αναφορά, ώστε η υπολογιζόμενη τιμή του να διατηρείται και έξω από τη συνάρτηση. Δεν μπορούμε απλά να επιστρέψουμε την τιμή του, καθώς χρειάζεται στο τέλος να επιστρέψουμε το φύλλο **leaf** στο οποίο έληξε η αναζήτηση. Αυτό είναι απαραίτητο, καθώς θα πρέπει να ξέρουμε σε ποιον κόμβο θα εφαρμοστούν οι πράξεις της ένθεσης και της διαγραφής, οι οποίες απαιτούν πρώτα ένα βήμα αναζήτησης στο δέντρο.

4.4.2 Ένθεση

Συνεχίζουμε βλέποντας πώς γίνεται η ένθεση νέων στοιχείων στα AVL δέντρα μας. Υλοποιούμε δύο είδη ένθεσης, μια για AVL ημερομηνιών που γίνεται με τη συνάρτηση **insertDateAVLnode** και μια για AVL θερμοκρασιών που γίνεται με τη συνάρτηση **insertTempAVLnode**. Και οι δύο αυτές συναρτήσεις αναλύονται στη συνέχεια.

4.4.2.1 Η συνάρτηση insertDateAVLnode

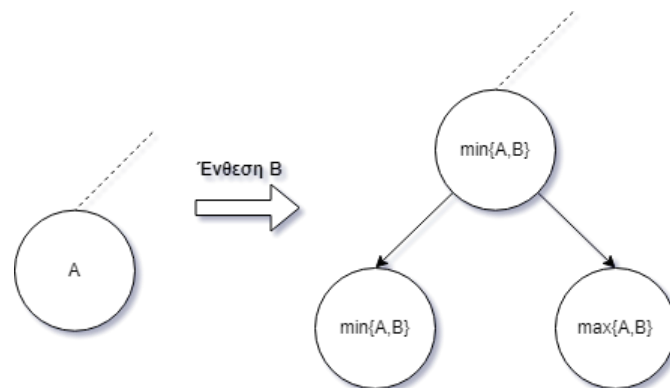
Δήλωση:

```
void insertDateAVLnode (AVLnode **tree, double temp, Date date)
```

Παράμετροι:

- **tree** – Το AVL ημερομηνιών μέσα στο οποίο θα κάνουμε την ένθεση.
- **date** – Η ημερομηνία του νέου κόμβου που θέλουμε να εισάγουμε.
- **temp** – Η θερμοκρασία του νέου κόμβου που θέλουμε να εισάγουμε.

Η συνάρτηση **insertDateAVLnode** ενθέτει στο AVL ημερομηνιών **tree** ένα νέο φύλλο με κλειδί την ημερομηνία **date** και πληροφορία τη θερμοκρασία **temp**. Η ένθεση γίνεται με τέτοιο τρόπο ώστε να μην υπάρχει ποτέ κόμβος με κενό παιδί, δηλαδή κάθε φορά εισάγουμε όχι έναν απλό κόμβο, αλλά ένα υποδέντρο που στα φύλλα περιέχει τον αρχικό και τον νέο κόμβο και στη ρίζα έχει το ελάχιστο από τα δύο στοιχεία:



Εικόνα 4.11 Ένθεση του B

Αρχικά, η συνάρτηση ελέγχει αν το δέντρο **tree** είναι κενό (**tree = NULL**). Σε αυτήν την περίπτωση, ο κόμβος που ενθέτουμε θα είναι η ρίζα του δέντρου, άρα πρέπει να αναθέσουμε στον δείκτη **tree** την τιμή του νέου κόμβου που δημιουργούμε (αυτός είναι και ο λόγος που το **tree** περνάει σαν διπλός δείκτης στη συνάρτηση):

```
if (*tree == NULL)
{
    *tree = (AVLnode *)malloc(sizeof(AVLnode));
    (*tree)->height = (*tree)->hb = 0;
    (*tree)->mult = 1;
    (*tree)->temperature = (double*)malloc(sizeof(double));
    (*tree)->temperature[0] = temp;
    (*tree)->date = (Date*)malloc(sizeof(Date));
    (*tree)->date[0] = date;
}
```

```
(*tree)->leftChild = (*tree)->rightChild = (*tree)->father = NULL;  
}
```

Μετά την αρχική ένθεση, θα ισχύει **tree** ≠ **NULL**. Τότε πρέπει να εκτελέσουμε μια αναζήτηση στο δέντρο, ώστε να προσδιορίσουμε τη θέση του νέου κόμβου:

```
AVLNode *foundNode = dateSearch(*tree, date, &found);
```

Στην περίπτωση όπου ο κόμβος υπάρχει ήδη, δεν τον δημιουργούμε πάλι, αλλά αυξάνουμε την πολλαπλότητά του και το μέγεθος του πίνακα θερμοκρασιών κατά 1 και αποθηκεύουμε τη νέα θερμοκρασία (όρισμα **temp**) στην τελευταία θέση του πίνακα:

```
foundNode->mult++;  
foundNode->temperature = (double*) realloc(foundNode->temperature, foundNode->mult * sizeof(double));  
foundNode->temperature[foundNode->mult-1] = temp;
```

Αν δεν υπάρχει, τότε δημιουργούμε δύο νέους κόμβους, οι οποίοι θα αποτελέσουν τα παιδιά του ήδη υπάρχοντος κόμβου, όπως φαίνεται στην Εικόνα 4.11. Ανάλογα με τη διάταξη της υπάρχουσας ημερομηνίας και της νέας ημερομηνίας (όρισμα **date**), οι τρεις κόμβοι αποκτούν τέτοιες τιμές, ώστε ο αριστερότερος μαζί με τον πατέρα του να έχουν τη μικρότερη ημερομηνία, ενώ ο δεξιότερος να έχει τη μεγαλύτερη.

Καθώς αντιγράφεται η πληροφορία του υπάρχοντος κόμβου σε ένα από τα δύο παιδιά, εννοείται πως αντιγράφεται και η πληροφορία σχετικά με την πολλαπλότητά του. Στην περίπτωση όπου ο υπάρχον κόμβος επίσης παραμένει και στη θέση του (δηλαδή είχε την ελάχιστη τιμή), τότε ο δείκτης με τις θερμοκρασίες απλά μεταφέρεται προς το αριστερό παιδί, ενώ ο ίδιος ο κόμβος, ο οποίος τώρα είναι εσωτερικός, αποκτά πολλαπλότητα 1 και έχει έναν νέο πίνακα με μόνο την πρώτη τιμή του πίνακα θερμοκρασιών. Αυτό στοχεύει στο να μη χρειάζεται να αντιγράψουμε κάθε φορά όλα τα περιεχόμενα του πίνακα με **memcpy**, το οποίο θα έπαιρνε πολύ χρόνο, αλλά ούτε και να αντιγράψουμε απλά τον δείκτη προς τον πίνακα, το οποίο θα δημιουργούσε προβλήματα στην περίπτωση που διαγράφαμε ένα φύλλο, αλλά ένας εσωτερικός κόμβος με το ίδιο διάνυσμα θερμοκρασιών υπήρχε ακόμα.

Τώρα που εισάχθηκε ένα νέο φύλλο στο δέντρο, τα ύψη και οι υψοζυγίσεις κάποιων κόμβων σίγουρα έχουν μεταβληθεί και χρειάζονται επαναυπολογισμό. Επίσης, υπάρχει περίπτωση η ένθεση να έχει διαταράξει τη δομή του δέντρου και αυτό να μην είναι πια ισοζυγισμένο. Για τα δύο αυτά ζητήματα φροντίζει η συνάρτηση **balanceFinder**, η οποία αναλύεται στην Ενότητα 4.5.

4.4.2.1 Η συνάρτηση insertTempAVLnode

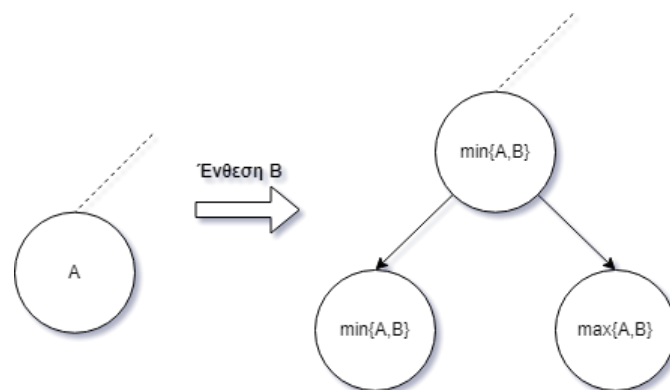
Δήλωση:

```
void insertTempAVLnode (AVLnode **tree, double temp, Date date)
```

Παράμετροι:

- **tree** – Το AVL θερμοκρασιών μέσα στο οποίο θα κάνουμε την ένθεση.
- **date** – Η ημερομηνία του νέου κόμβου που θέλουμε να εισάγουμε.
- **temp** – Η θερμοκρασία του νέου κόμβου που θέλουμε να εισάγουμε.

Η συνάρτηση **insertTempAVLnode** ενθέτει στο AVL θερμοκρασιών tree ένα νέο φύλλο με κλειδί την θερμοκρασία **temp** και πληροφορία την ημερομηνία **date**. Η ένθεση γίνεται με τέτοιο τρόπο ώστε να μην υπάρχει ποτέ κόμβος με κενό παιδί, δηλαδή κάθε φορά εισάγουμε όχι έναν απλό κόμβο, αλλά ένα υποδέντρο που στα φύλλα περιέχει τον αρχικό και τον νέο κόμβο και στη ρίζα έχει το ελάχιστο από τα δύο στοιχεία:



Εικόνα 4.12 Ένθεση του B

Αρχικά, η συνάρτηση ελέγχει αν το δέντρο tree είναι κενό (**tree = NULL**). Σε αυτήν την περίπτωση, ο κόμβος που ενθέτουμε θα είναι η ρίζα του δέντρου, άρα πρέπει να αναθέσουμε στον δείκτη **tree** την τιμή του νέου κόμβου που δημιουργούμε (αυτός είναι και ο λόγος που το **tree** περνάει σαν διπλός δείκτης στη συνάρτηση):

```
if (*tree == NULL)
{
    *tree = (AVLnode *)malloc(sizeof(AVLnode));
    (*tree)->height = (*tree)->hb = 0;
    (*tree)->mult = 1;
    (*tree)->temperature = (double*)malloc(sizeof(double));
    (*tree)->temperature[0] = temp;
    (*tree)->date = (Date*)malloc(sizeof(Date));
    (*tree)->date[0] = date;
}
```

```

    (*tree)->leftChild = (*tree)->rightChild = (*tree)->father = NULL;
}

```

Μετά την αρχική ένθεση, θα ισχύει **tree** ≠ **NULL**. Τότε πρέπει να εκτελέσουμε μια αναζήτηση στο δέντρο, ώστε να προσδιορίσουμε τη θέση του νέου κόμβου:

```
AVLnode *foundNode = tempSearch(*tree, temp, &found);
```

Στην περίπτωση όπου ο κόμβος υπάρχει ήδη, δεν τον δημιουργούμε πάλι, αλλά αυξάνουμε την πολλαπλότητά του και το μέγεθος του πίνακα ημερομηνιών κατά 1 και αποθηκεύουμε τη νέα ημερομηνία (όρισμα **date**) στην τελευταία θέση του πίνακα:

```

foundNode->mult++;
foundNode->date      =      (Date*) realloc(foundNode->date, foundNode->
>mult*sizeof(Date));
foundNode->date[foundNode->mult-1] = date;

```

Αν δεν υπάρχει, τότε δημιουργούμε δύο νέους κόμβους, οι οποίοι θα αποτελέσουν τα παιδιά του ήδη υπάρχοντος κόμβου, όπως φαίνεται στην Εικόνα 4.12. Ανάλογα με τη διάταξη της υπάρχουσας θερμοκρασίας και της νέας θερμοκρασίας (όρισμα **temp**), οι τρεις κόμβοι αποκτούν τέτοιες τιμές, ώστε ο αριστερότερος μαζί με τον πατέρα του να έχουν τη μικρότερη θερμοκρασία, ενώ ο δεξιότερος να έχει τη μεγαλύτερη.

Καθώς αντιγράφεται η πληροφορία του υπάρχοντος κόμβου σε ένα από τα δύο παιδιά, εννοείται πώς αντιγράφεται και η πληροφορία σχετικά με την πολλαπλότητά του. Στην περίπτωση όπου ο υπάρχον κόμβος επίσης παραμένει και στη θέση του (δηλαδή είχε την ελάχιστη τιμή), τότε ο δείκτης με τις ημερομηνίες απλά μεταφέρεται προς το αριστερό παιδί, ενώ ο ίδιος ο κόμβος, ο οποίος τώρα είναι εσωτερικός, αποκτά πολλαπλότητα 1 και έχει έναν νέο πίνακα με μόνο την πρώτη τιμή του πίνακα ημερομηνιών. Αυτό στοχεύει στο να μη χρειάζεται να αντιγράφουμε κάθε φορά όλα τα περιεχόμενα του πίνακα με **memcpy**, το οποίο θα έπαιρνε πολύ χρόνο, αλλά ούτε και να αντιγράφουμε απλά τον δείκτη προς τον πίνακα, το οποίο θα δημιουργούσε προβλήματα στην περίπτωση που διαγράφαμε ένα φύλλο, αλλά ένας εσωτερικός κόμβος με το ίδιο διάνυσμα ημερομηνιών υπήρχε ακόμα.

Τώρα που εισάχθηκε ένα νέο φύλλο στο δέντρο, τα ύψη και οι υψοζυγίσεις κάποιων κόμβων σίγουρα έχουν μεταβληθεί και χρειάζονται επαναυπολογισμό. Επίσης, υπάρχει περίπτωση η ένθεση να έχει διαταράξει τη δομή του δέντρου και αυτό να μην είναι πια ισοζυγισμένο. Για τα δύο αυτά ζητήματα φροντίζει η συνάρτηση **balanceFinder**, η οποία αναλύεται στην Ενότητα 4.5.

4.4.3 Διαγραφή

Τελειώνουμε με τις πράξεις πάνω στα δέντρα βλέποντας πώς γίνεται η διαγραφή κόμβων. Σε αντίθεση με την αναζήτηση και την ένθεση, εδώ υλοποιούμε ένα μόνο είδος διαγραφής για τα AVL ημερομηνιών, καθώς δε χρειάζεται να διαγράψουμε κόμβους στην περίπτωση των AVL θερμοκρασιών. Η διαγραφή γίνεται με τη συνάρτηση **deleteAVLnode**, την οποία αναλύουμε στη συνέχεια.

4.4.3.1 Η συνάρτηση deleteAVLnode

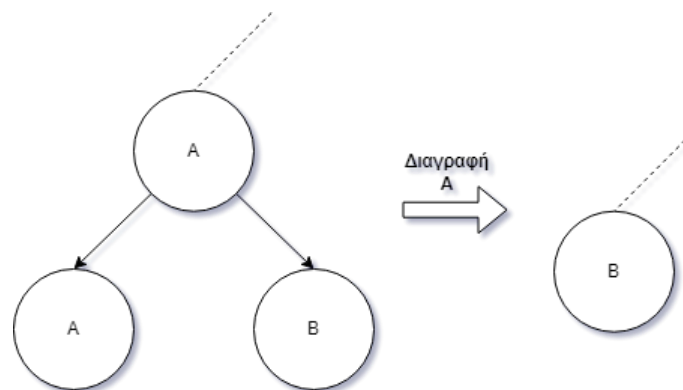
Δήλωση:

```
void deleteAVLnode (AVLnode **tree, Date date)
```

Παράμετροι:

- **tree** – Το AVL ημερομηνιών μέσα στο οποίο θα κάνουμε τη διαγραφή.
- **date** – Την ημερομηνία που θα διαγράψουμε.

Η συνάρτηση **deleteAVLnode** διαγράφει στο AVL ημερομηνιών **tree** ένα φύλλο με κλειδί την ημερομηνία **date**. Με τη διαγραφή του φύλλου, διαγράφεται και το αντίστοιχο κλειδί του πατέρα. Επειδή όμως βρισκόμαστε σε δυαδικό δέντρο, όπου κάθε κόμβος έχει ένα μόνο κλειδί, αυτό έχει ως αποτέλεσμα τη διαγραφή του πατέρα, του οποίου τη θέση παίρνει το παιδί που δε διαγράψαμε:



Εικόνα 4.13 Διαγραφή του A

Αρχικά εκτελούμε μια αναζήτηση στο δέντρο, ώστε να βρούμε τη θέση του φύλλου που πρέπει να διαγραφεί:

```
AVLnode *foundNode = dateSearch(*tree, date, &found);
```

Αν το φύλλο δε βρεθεί, δεν κάνουμε τίποτα. Στην περίπτωση που βρέθηκε, μπορεί να αρχίσει η διαδικασία της διαγραφής. Η διαγραφή εμπλέκει τρεις κόμβους: το φύλλο που θα διαγραφεί, τον πατέρα του και τον παππού του. Θα πρέπει, επομένως, να λάβουμε υπ' όψη μας τρεις διαφορετικές περιπτώσεις, σε περίπτωση που κάποιος απ' αυτούς δεν υπάρχει.

Αρχικά, αν δεν υπάρχει ο πατέρας, σημαίνει πως το φύλλο που θα διαγράψουμε είναι και η ρίζα του δέντρου. Σε αυτήν την περίπτωση, απλά διαγράφουμε τον κόμβο και αναθέτουμε στον δείκτη **tree** την τιμή **NULL** (αυτός είναι και ο λόγος που το **tree** περνάει σαν διπλός δείκτης στη συνάρτηση):

```
// if only the root is left
if (nodeFather == NULL)
{
    free(foundNode->temperature);
    free(foundNode->date);
    free(foundNode);
    *tree = NULL;
    printf("\nEntry deleted successfully and the tree is now empty\n");
    return;
}
```

Στην περίπτωση που υπάρχει πατέρας, αλλά όχι παππούς (δηλαδή ο πατέρας είναι η ρίζα), τότε διαγράφουμε το φύλλο και τη ρίζα και η νέα ρίζα του δέντρου γίνεται το φύλλο που δεν διαγράψαμε:

```
if (foundNode == nodeFather->leftChild)
{
    *tree = nodeFather->rightChild;
    nodeFather->rightChild->father = NULL;
}
else
{
    *tree = nodeFather->leftChild;
    nodeFather->leftChild->father = NULL;
}
```

Στη γενική περίπτωση όπου υπάρχει και πατέρας και παππούς, τότε διαγράφουμε φύλλο και πατέρα και αλλάζουμε τον δείκτη από τον παππού στον πατέρα, ώστε τώρα να δείχνει στο φύλλο που δε διαγράψαμε (το οποίο αποκαλούμε **nodeBro**):

```
if (foundNode == nodeFather->leftChild)
    nodeBro = nodeFather->rightChild;
else if (foundNode == nodeFather->rightChild)
    nodeBro = nodeFather->leftChild;
```



```
nodeBro->father = nodeGrandpa;

if (nodeFather == nodeGrandpa->leftChild)
    nodeGrandpa->leftChild = nodeBro;
else if (nodeFather == nodeGrandpa->rightChild)
    nodeGrandpa->rightChild = nodeBro;
```

Τώρα που διαγράφηκε ένα φύλλο από το δέντρο, τα ύψη και οι υψοζυγίσεις κάποιων κόμβων σίγουρα έχουν μεταβληθεί και χρειάζονται επαναυπολογισμός. Επίσης, υπάρχει περίπτωση η διαγραφή να έχει διαταράξει τη δομή του δέντρου και αυτό να μην είναι πια ισοζυγισμένο. Για τα δύο αυτά ζητήματα φροντίζουν οι συναρτήσεις **balanceFinder** και **deleteRotation** αντίστοιχα, οι οποίες αναλύονται στην Ενότητα 4.5.

Ενότητα 4.5 Υπολογισμός ύψους και υψοζύγισης

Σε αυτήν την ενότητα θα δούμε πώς υπολογίζουμε το νέο ύψος και τη νέα υψοζύγιση των κόμβων ύστερα από μια πράξη στο δέντρο. Θα δούμε επίσης πότε χρειάζεται να εφαρμοστούν διορθωτικές πράξεις σε κάποιον κόμβο, ώστε να επαναφέρουμε τη δομή του AVL δέντρου.

4.5.1 Η συνάρτηση `balanceFinder`

Δήλωση:

```
AVLnode *balanceFinder(AVLnode *foundNode, int functionUse)
```

Παράμετροι:

- **foundNode** – Ο κόμβος από τον οποίο θα ξεκινήσει ο υπολογισμός του ύψους και της υψοζύγισης.
- **functionUse** – Καθορίζει αν η πράξη που έγινε ήταν ένθεση ή διαγραφή.

Επιστρέφει:

Τη ρίζα του υποδέντρου μετά από κάποια τερματική περιστροφή.

Η συνάρτηση **balanceFinder** δέχεται ως όρισμα έναν κόμβο και ξεκινά μια επαναληπτική διαδικασία επαναυπολογισμού των υψών και των υψοζυγίσεων όλων των προγόνων. Η διαδικασία αυτή σταματά μόλις φτάσουμε στη ρίζα ή μόλις γίνει τερματική περιστροφή. Η συνάρτηση εκτός από τα παραπάνω εκτελεί και τις απαραίτητες περιστροφές για τη διόρθωση του δέντρου, **αλλά μόνο στην περίπτωση των ενθέσεων**. Διακρίνουμε μεταξύ ένθεσης και διαγραφής με το όρισμα **functionUse**, όπου

- Για **functionUse = 1** έχουμε ένθεση.

- Για **functionUse** $\neq 1$ έχουμε διαγραφή.

Συνήθως περνάμε ως όρισμα έναν εσωτερικό κόμβο που είναι πατέρας φύλλου, άρα **height** = 1 και **hb** = 0:

```
AVLnode *currentNode = foundNode;
currentNode->height = 1;
currentNode->hb = 0;
```

Αν όμως τύχει να περάσουμε ως όρισμα ένα φύλλο (το οποίο συμβαίνει στην περίπτωση των διαγραφών), θέτουμε **height** και **hb** ίσα με το 0:

```
if (currentNode->leftChild == NULL) //Used in deletions, because we
pass the leaf as the argument (unlike insertions)
{
    currentNode->height = 0;
    currentNode->hb = 0;
}
```

Εδώ πρέπει να επισημάνουμε μια πολύ σημαντική διαφορά στον τρόπο που επιλέξαμε να διαχειριστούμε την ένθεση και τη διαγραφή. Στην ένθεση η κλήση της **balanceFinder** γίνεται μετά την προσθήκη του νέου κόμβου, ενώ στη διαγραφή η κλήση της γίνεται πριν την αφαίρεση του κόμβου. Γι' αυτό και στην ένθεση το όρισμα είναι εσωτερικός κόμβος, ενώ στη διαγραφή είναι φύλλο.

Στη συνέχεια υπολογίζουμε το ύψος και την υψοζύγιση για τον εξεταζόμενο κόμβο σύμφωνα με τις σχέσεις:

height = **max**{**leftChild**->**height**, **rightChild**->**height**} + 1

και

hb = **rightChild**->**height** - **leftChild**->**height**:

```
currentNode->height = max(currentNode->leftChild,    currentNode->
>rightChild) + 1;

currentNode->hb = currentNode->rightChild->height - currentNode->
>leftChild->height;
```

Στην περίπτωση όπου είχαμε ένθεση, το επόμενο βήμα είναι να εφαρμόσουμε τις κατάλληλες περιστροφές, ώστε να διορθώσουμε τη δομή του δέντρου. Υπάρχουν 5 περιπτώσεις που πρέπει να καλύψουμε:

- Αν **currentNode->hb** = 0, τότε γίνεται απορρόφηση και δε χρειάζεται περιστροφή:

```
if (currentNode->hb == 0)
{
    newRoot = currentNode;
    break;
}
```

- Αν **currentNode->hb** = +2 και **currentNode->rightChild->hb** = +1, τότε είμαστε στην περίπτωση RR και χρειάζεται αριστερή περιστροφή γύρω από τον **currentNode**:

```
if (currentNode->hb == 2)
{
    if (originNode->hb == 1)
        newRoot = leftRotation(currentNode);
    .
    .
    .
    break;
}
```

- Αν **currentNode->hb** = +2 και **currentNode->rightChild->hb** = -1, τότε είμαστε στην περίπτωση RL και χρειάζεται αριστερή-δεξιά περιστροφή γύρω από τον **currentNode**:

```
if (currentNode->hb == 2)
{
    .
    .
    .
    else if (originNode->hb == -1)
        newRoot = leftRightRotation(currentNode);
    break;
}
```

- Αν **currentNode->hb** = -2 και **currentNode->leftChild->hb** = -1, τότε είμαστε στην περίπτωση LL και χρειάζεται δεξιά περιστροφή γύρω από τον **currentNode**:

```
if (currentNode->hb == -2)
{
    if (originNode->hb == -1)
        newRoot = rightRotation(currentNode);
    .
    .
    .
    break;
}
```

```
}
```

- Αν **currentNode->hb** = -2 και **currentNode->leftChild->hb** = +1, τότε είμαστε στην περίπτωση LR και χρειάζεται δεξιά-αριστερή περιστροφή γύρω από τον **currentNode**.

```
if (currentNode->hb == -2)
{
    .
    .
    .
    else if (originNode->hb == 1)
        newRoot = rightLeftRotation(currentNode);

    break;
}
```

Κάθε μια απ' αυτές τις περιστροφές είναι τερματικές για την ένθεση, καθώς δεν μεταβάλλουν το ύψος του υποδέντρου. Επομένως, αποθηκεύουμε το αποτέλεσμα τους (δηλαδή τη νέα ρίζα του υποδέντρου στο οποίο έγινε περιστροφή) στη μεταβλητή **newRoot**, την οποία επιστρέφουμε από την **balanceFinder** στην **insertAVLnode**, ώστε άμα η ρίζα του υποδέντρου είναι και η ρίζα όλου του δέντρου, να αλλάξει η τιμή του δείκτη προς τη ρίζα:

```
return newRoot;
```

4.5.2 Η συνάρτηση deleteRotation

Δήλωση:

```
void deleteRotation(AVLnode *node1, AVLnode *node2, AVLnode
**newRoot)
```

Παράμετροι:

- **node1** – Ο παππούς του κόμβου που διαγράφηκε.
- **node2** – Ο αδερφός του κόμβου.
- **newRoot** – Το δέντρο στο οποίο έγινε η διαγραφή.

Η συνάρτηση **deleteRotation** είναι αντίστοιχη της **balanceFinder**, αλλά για την περίπτωση των διαγραφών. Δέχεται ως ορίσματα δύο κόμβους, το φύλλο που δε διαγράφηκε **node2** και τον πατέρα του **node1** και ξεκινά μια επαναληπτική διαδικασία επαναυπολογισμού των υψών και των υψοζυγίσεων όλων των προγόνων:

```
while (nodeGrandpa != NULL)
```

```

{
    .
    .
    .
    nodePrev->height = max(nodePrev->leftChild, nodePrev->
>rightChild) + 1;

    nodePrev->hb = nodePrev->rightChild->height - nodePrev->
>leftChild->height;
    .
    .
    .
    nodePrev = nodeGrandpa;
    nodeGrandpa = nodeGrandpa->father;
}

```

Η διαδικασία συνεχίζεται μέχρι να βρεθεί κρίσιμος κόμβος απόσβεσης, όπου κάνουμε **break** από το **while** και έπειτα ελέγχουμε σε ποια περίπτωση βρισκόμαστε. Εδώ να σημειώσουμε ότι σε αντίθεση με την ένθεση, στη διαγραφή επιλέξαμε να ελέγχουμε σε ποια περίπτωση βρισκόμαστε χρησιμοποιώντας την τιμή της υψοζύγισης του πατέρα PPIN αυτή επαναυπολογιστεί. Επομένως:

- Αν **nodeGrandpa->hb** = 0, τότε γίνεται απορρόφηση και δε χρειάζεται περιστροφή. Η περίπτωση αυτή είναι τερματική. Αν δεν είμαστε σ' αυτήν την περίπτωση, θέτουμε για τις επόμενες περιπτώσεις ως **nodeKid** το παιδί του **nodeGrandpa** που δεν βρίσκεται στο μονοπάτι απόσβεσης (δηλαδή το άλλο παιδί, αυτό με το μεγαλύτερο ύψος).

```

if (nodeGrandpa->hb == 0)
{
    nodeGrandpa->height = max(nodeGrandpa->leftChild,
nodeGrandpa->rightChild) + 1;
    nodeGrandpa->hb = nodeGrandpa->rightChild->height -
nodeGrandpa->leftChild->height;

    return;
}

```

- Αν **nodeGrandpa->hb** = +1 και **nodeKid->hb** = 0, τότε απλά γίνεται μια τερματική αριστερή περιστροφή γύρω από τον **nodeGrandpa**.
- Αν **nodeGrandpa->hb** = +1 και **nodeKid->hb** = +1, τότε είμαστε στην περίπτωση RR και χρειάζεται αριστερή περιστροφή γύρω από τον **nodeGrandpa**. Η περίπτωση αυτή δεν είναι τερματική, άρα καλούμε πάλι τη συνάρτηση.
- Αν **nodeGrandpa->hb** = +1 και **nodeKid->hb** = -1, τότε είμαστε στην περίπτωση RL και χρειάζεται αριστερή-δεξιά περιστροφή γύρω από τον **nodeGrandpa**. Η περίπτωση αυτή δεν είναι τερματική, άρα καλούμε πάλι τη συνάρτηση.

```

if (nodeGrandpa->hb == 1)
{
    .
    .
    .

    if (nodeKid->hb == 1)
    {
        *newRoot = leftRotation(nodeGrandpa);
        deleteRotation((*newRoot)->father, *newRoot, newRoot);
    }
    else if (nodeKid->hb == -1)
    {
        *newRoot = leftRightRotation(nodeGrandpa);
        deleteRotation((*newRoot)->father, *newRoot, newRoot);
    }
    else
        *newRoot = leftRotation(nodeGrandpa);
}

```

- Αν **nodeGrandpa->hb** = -1 και **nodeKid->hb** = 0, τότε απλά γίνεται μια τερματική δεξιά περιστροφή γύρω από τον **nodeGrandpa**.
- Αν **nodeGrandpa->hb** = -1 και **nodeKid->hb** = +1, τότε είμαστε στην περίπτωση LR και χρειάζεται δεξιά-αριστερή περιστροφή γύρω από τον **nodeGrandpa**. Η περίπτωση αυτή δεν είναι τερματική, άρα καλούμε πάλι τη συνάρτηση.
- Αν **nodeGrandpa->hb** = -1 και **nodeKid->hb** = -1, τότε είμαστε στην περίπτωση LL και χρειάζεται δεξιά περιστροφή γύρω από τον **nodeGrandpa**. Η περίπτωση αυτή δεν είναι τερματική, άρα καλούμε πάλι τη συνάρτηση.

```

if (nodeGrandpa->hb == -1)
{
    .
    .
    .

    if (nodeKid->hb == -1)
    {
        *newRoot = rightRotation(nodeGrandpa);
        deleteRotation((*newRoot)->father, *newRoot, newRoot);
    }
    else if (nodeKid->hb == 1)
    {
        *newRoot = rightLeftRotation(nodeGrandpa);
        deleteRotation((*newRoot)->father, *newRoot, newRoot);
    }
    else
        *newRoot = rightRotation(nodeGrandpa);
}

```

Το αποτέλεσμα κάθε περιστροφής αποθηκεύεται στη μεταβλητή **newRoot**, ώστε να ελέγχουμε έξω απ' τη συνάρτηση αν πρέπει να αλλάξουμε τη ρίζα του δέντρου:

```
if (newRoot->father == NULL)
    *tree = newRoot;
```

4.5.3 Η συνάρτηση **max**

Δήλωση:

```
int max(AVLnode *node1, AVLnode *node2)
```

Παράμετροι:

- **node1** – Ο πρώτος κόμβος που εξετάζουμε.
- **node2** – Ο δεύτερος κόμβος που εξετάζουμε.

Επιστρέφει:

Το μεγαλύτερο από τα ύψη των κόμβων **node1** και **node2**.

Η συνάρτηση **max** δέχεται ως ορίσματα δύο κόμβους **node1** και **node2** και επιστρέφει το μεγαλύτερο από τα ύψη τους. Χρησιμοποιείται όταν θέλουμε να υπολογίσουμε το ύψος του πατέρα τους, το οποίο υπολογίζεται από την παρακάτω σχέση:

```
max{leftChild->height, rightChild->height} + 1
```

Ενότητα 4.6 Άλλες συναρτήσεις

Σε αυτήν την ενότητα εξετάζουμε άλλες συναρτήσεις τις οποίες χρειάστηκε να υλοποιήσουμε στο πρόγραμμά μας.

4.6.1 Η συνάρτηση `inOrder`

Δήλωση:

```
void inOrder (AVLnode *tree)
```

Παράμετροι:

- **tree** – Το δέντρο στο οποίο θα εφαρμόσουμε ενδοδιατεταγμένη διαπέραση.

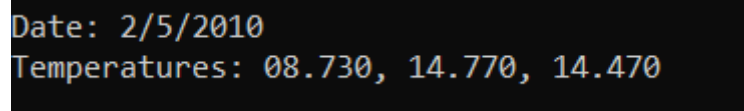
Η συνάρτηση `inOrder` διαπερνά με αναδρομικό τρόπο το μη κενό δέντρο **tree**, ξεκινώντας με το αριστερό υποδέντρο, έπειτα εκτυπώνοντας τη ρίζα και καταλήγοντας στο δεξιό υποδέντρο:

```
inOrder (tree->leftChild);
```

```
(εκτύπωση ρίζας)
```

```
inOrder (tree->rightChild);
```

Κατά την εκτύπωση ενός κόμβου, εκτυπώνουμε την ημερομηνία, καθώς επίσης και κάθε θερμοκρασία του αρχείου **ocean.csv** που αντιστοιχεί στην ημερομηνία αυτή:



```
Date: 2/5/2010
Temperatures: 08.730, 14.770, 14.470
```

Εικόνα 4.14 Το αποτέλεσμα της εκτύπωσης για ένα τυχαίο φύλλο

Να σημειωθεί ότι, παρόλο που στο αρχείο **ocean.csv** οι ημερομηνίες είναι στη μορφή MM/DD/YYYY, εμείς στο πρόγραμμά μας τις εκτυπώνουμε ως DD/MM/YYYY.

4.6.2 Η συνάρτηση minTemp

Δήλωση:

```
void minTemp (AVLnode* tree)
```

Παράμετροι:

- **tree** – Το AVL θερμοκρασιών για το οποίο θέλουμε να βρούμε ελάχιστη θερμοκρασία.

Η συνάρτηση **minTemp** βρίσκει την ελάχιστη θερμοκρασία που είναι αποθηκευμένη σε ένα AVL θερμοκρασιών και εκτυπώνει όλες τις ημερομηνίες όπου αυτή συναντάται. Για να βρούμε τον κόμβο με την ελάχιστη θερμοκρασία, αρκεί να ξεκινήσουμε από τη ρίζα και να ακολουθούμε διαρκώς το αριστερό παιδί του κάθε κόμβου, μέχρι τελικά να φτάσουμε σε ένα φύλλο:

```
while (node->leftChild != NULL)
    node = node->leftChild;
```

```
=====
Minimum temperature of 07.600 found in 1 dates:
30/4/2010
=====
```

Εικόνα 4.15 Εύρεση ελάχιστης θερμοκρασίας στο ocean.csv

Να σημειωθεί ότι, παρόλο που στο αρχείο **ocean.csv** οι ημερομηνίες είναι στη μορφή MM/DD/YYYY, εμείς στο πρόγραμμά μας τις εκτυπώνουμε ως DD/MM/YYYY.

4.6.3 Η συνάρτηση maxTemp

Δήλωση:

```
void maxTemp (AVLnode* tree)
```

Παράμετροι:

- **tree** – Το AVL θερμοκρασιών για το οποίο θέλουμε να βρούμε μέγιστη θερμοκρασία.

Η συνάρτηση **maxTemp** βρίσκει τη μέγιστη θερμοκρασία που είναι αποθηκευμένη σε ένα AVL θερμοκρασιών και εκτυπώνει όλες τις ημερομηνίες όπου αυτή συναντάται. Για να βρούμε τον κόμβο με τη μέγιστη θερμοκρασία, αρκεί να ξεκινήσουμε από τη ρίζα και να ακολουθούμε διαρκώς το δεξιό παιδί του κάθε κόμβου, μέχρι τελικά να φτάσουμε σε ένα φύλλο:

```
while (node->rightChild != NULL)
    node = node->rightChild;
```

```
=====
Maximum temperature of 25.420 found in 4 dates:
26/10/2018
25/10/2018
13/4/2019
10/4/2019
=====
```

Εικόνα 4.16 Εύρεση μέγιστης θερμοκρασίας στο ocean.csv

Να σημειωθεί ότι, παρόλο που στο αρχείο **ocean.csv** οι ημερομηνίες είναι στη μορφή MM/DD/YYYY, εμείς στο πρόγραμμά μας τις εκτυπώνουμε ως DD/MM/YYYY.

Κεφάλαιο 5

Κατακερματισμός

Στο κεφάλαιο αυτό θα δούμε έναν εναλλακτικό τρόπο για να επεξεργαζόμαστε τα δεδομένα του αρχείου `ocean.csv` χρησιμοποιώντας μια δομή hashing

Ενότητα 5.1 Αναπαράσταση των δεδομένων

Όπως στα AVL, έτσι και εδώ χρειαζόμαστε έναν τρόπο να αποθηκεύουμε τα δεδομένα του αρχείου **ocean.csv**, αυτή τη φορά σε μια δομή hashing. Υπενθυμίζουμε ότι το αρχείο **ocean.csv** αποτελείται από γραμμές, όπου κάθε γραμμή-δείγμα αντιστοιχεί σε μια μέτρηση της θερμοκρασίας (σε °C) στο νερό του ωκεανού καθώς και των τιμών Phosphate, Silicate, Nitrite, Nitrate, Salinity, Oxygen για την περίοδο 2000 έως 2019. Η δομή κάθε γραμμής είναι:

Date	Temp	Phosphate	Silicate	Nitrite	Nitrate	Salinity	Oxygen
------	------	-----------	----------	---------	---------	----------	--------

Εικόνα 5.1 Δομή μιας γραμμής του `ocean.csv`

Στην περίπτωση μας μάς ενδιαφέρουν μόνο τα πεδία `Date` και `Temp`.

Ο κατακερματισμός χωρίζει με βάση την συνάρτηση κατακερματισμού τα στοιχεία σε διαφορετικές διασυνδεδεμένες λίστες. Για την υλοποίηση αυτών των λιστών ορίζουμε δύο διαφορετικές δομές τις `Node` και `Bucket`.

```
typedef struct node{
    struct node * next;
    Date date;
    float temp;
}
Node;

typedef struct{
    Node * first;
    Node * last;
}
Bucket;
```

Κάθε **Node** αναπαριστά την εγγραφή μιας γραμμής του **ocean.csv**, διατηρώντας μόνο την ημερομηνία και την θερμοκρασία στα πεδία **date** και **temp** αντίστοιχα, και έχει και ένα επιπλέον πεδίο δείκτη **next** που δείχνει στο επόμενο στοιχείο της λίστας.

Κάθε **Bucket** αναπαριστά την κάθε διασυνδεδεμένη λίστα και περιέχει δύο δείκτες οι οποίοι δείχνουν το πρώτο και τελευταίο στοιχείο της λίστας αντίστοιχα.

Ενότητα 5.2 Συνάρτηση κατακερματισμού

Στην ενότητα αυτή θα δούμε τις συναρτήσεις που χρησιμοποιούμε για την υλοποίηση της συνάρτησης κατακερματισμού. Το κλειδί κάθε στοιχείου είναι το string της ημερομηνίας του. Ο κουβάς στον οποίο αντιστοιχεί στοιχείο με string ημερομηνίας *str*, είναι το υπόλοιπο της διαίρεσης του αθροίσματος των τιμών ASCII των χαρακτήρων του *str* με την σταθερά **BUCKETS**. Για τον υπολογισμό αυτό χρησιμοποιούμε τις συναρτήσεις **getKey** και **hash**.

5.2.1 Η συνάρτηση getKey

Δήλωση:

```
char* getKey(Date date)
```

Παράμετροι:

- **date** – Η ημερομηνία που θα μετατραπεί σε string.

Επιστρέφει:

Το αλφαριθμητικό που αντιστοιχεί στην ημερομηνία **date**

Η συνάρτηση **getKey** δέχεται ως ορίσματα μια ημερομηνία **date** και την μετατρέπει σε ένα αλφαριθμητικό της μορφής "YYYY-MM-DD". Το αλφαριθμητικό αυτό χρησιμοποιείται στη συνέχεια ως κλειδί για την αναζήτηση μιας ημερομηνίας στον πίνακα κατακερματισμού. Η αντιστοίχιση του αλφαριθμητικού σε θέση πίνακα γίνεται με τη συνάρτηση **hash**

5.2.2 Η συνάρτηση `hash`

Δήλωση:

```
int hash(char* key)
```

Παράμετροι:

- **key** – Το κλειδί-όρισμα της συνάρτησης κατακερματισμού.

Επιστρέφει:

Η θέση στον πίνακα κατακερματισμού που αντιστοιχεί στο κλειδί `key`

Η συνάρτηση **hash** αντιστοιχίζει μια ημερομηνία που έχει μετατραπεί σε αλφαριθμητικό με τη συνάρτηση **getKey** στη θέση του πίνακα κατακερματισμού όπου θα γίνει η αποθήκευσή της. Η μετατροπή γίνεται σύμφωνα με την παρακάτω συνάρτηση κατακερματισμού:

$$[\sum_i ASCII(key[i])] \bmod BUCKETS$$

Ενότητα 5.3 Αποθήκευση των δεδομένων

Η μετατροπή των δεδομένων γίνεται με παρόμοιο τρόπο με αυτόν που παρουσιάσαμε στο Κεφάλαιο 1. Όπως και πριν, θα χρειαστεί να διαβάζουμε κάθε γραμμή του **ocean.csv** και να εξάγουμε την πληροφορία που χρειαζόμαστε.

Αφού διαβάσουμε το αρχείο και δημιουργήσουμε ένα array με όλα τα στοιχεία, δεσμεύουμε χώρο για τα Buckets του πίνακα κατακερματισμού **hashtable**, το πλήθος των οποίων καθορίζεται από την σταθερά **BUCKETS**. Στην συνέχεια γεμίζουμε τα Buckets κάνοντας χρήση των συναρτήσεων **insert** και **insertAll**, οι οποίες αναλύονται παρακάτω.

5.3.1 Η συνάρτηση `insert`

Δήλωση:

```
void insert(Data data, Bucket* bucket)
```

Παράμετροι:

- **data** – Τα δεδομένα στα οποία βασίζεται η εγγραφή που θα δημιουργηθεί.
- **bucket** – Δείκτης στον κουβά όπου θα προστεθεί η νέα εγγραφή.

Η συνάρτηση `insert` δημιουργεί μια εγγραφή με τα πεδία **date** και **temp** του **data**, και την προσθέτει στο τέλος του **bucket** κάνοντας το πεδίο **next** του τελευταίου **Node** του **bucket** το πεδίο **last** του **bucket** να δείχνουν στη νέα εγγραφή. Η νέα εγγραφή έχει αρχικά πεδίο **next** με τιμή **NULL**.

5.3.2 Η συνάρτηση `insertAll`

Δήλωση:

```
void insertAll(Data* data, int maxsize, Bucket* hashtable)
```

Παράμετροι:

- **data** – Ο πίνακας που περιέχει όλα τα στοιχεία που θα μετατραπούν σε εγγραφές.
- **maxsize** – Ο μέγιστος αριθμός θέσης όλου του πίνακα **data**.
- **hashtable** – Ο πίνακας κατακερματισμού στον οποίο θα προστεθούν όλες οι εγγραφές.

Η συνάρτηση `insertAll` δημιουργεί έναν πίνακα κατακερματισμού και τον αποθηκεύει στη μεταβλητή **hashtable**. Με την χρήση βρόγχου υπολογίζει για κάθε στοιχείο σε ποιον κουβά πρέπει να δημιουργηθεί η αντίστοιχη εγγραφή του και καλεί την `insert` για την δημιουργία της εγγραφής.

Ενότητα 5.4 Υλοποίηση πράξεων

Στο πρόγραμμα έχουμε υλοποιήσει τις πράξεις `search`, `edit`, `delete` με συναρτήσεις που αναλύονται παρακάτω.

5.4.1 Η συνάρτηση `searchNode`

Δήλωση:

```
Node* searchNode(Date date, Bucket* hashtable)
```

Παράμετροι:

- **date** – Η ημερομηνία που αναζητάμε.
- **hashtable** – Ο πίνακας κατακερματισμού στον οποίο θα εκτελεστεί η αναζήτηση.

Επιστρέφει:

Δείκτη στο πρώτο **Node** που θα βρεθεί με την ημερομηνία **date**. Αν δεν βρεθεί, επιστρέφει **NULL**.

Η συνάρτηση **searchNode** εκτελεί την πράξη της αναζήτησης για τον πίνακα κατακερματισμού. Αρχικά υπολογίζει το **index** του κουβά στον οποίο αντιστοιχεί η ημερομηνία **date** και στην συνέχεια εκτελεί μια απλή γραμμική αναζήτηση.

5.4.2 Η συνάρτηση `editNode`

Δήλωση:

```
void editNode(Node* node, float temp)
```

Παράμετροι:

- **node** – Ο κόμβος που θα τροποποιηθεί.
- **temp** – Η νέα τιμή του πεδίου **temp**.

Η συνάρτηση **editNode** δέχεται ως ορίσματα έναν κόμβο **node** και αποθηκεύει σε αυτόν τη θερμοκρασία που δίνεται από το όρισμα **temp**:

```
node->temp = temp;
```

5.4.3 Η συνάρτηση deleteNode

Δήλωση:

```
void deleteNode(Data date, Bucket* hashtable)
```

Παράμετροι:

- **date** – Η ημερομηνία του πεδίου που θέλουμε να διαγράψουμε.
- **hashtable** – Ο πίνακας κατακερματισμού στον οποίο θα γίνει η αναζήτηση και η διαγραφή.

Η συνάρτηση **deleteNode** διαγράφει έναν κόμβο από τον πίνακα κατακερματισμού. Για την διαγραφή ενός κόμβου πρέπει να αλλάξουμε την τιμή του δείκτη **next** του προηγούμενου στοιχείου. Έτσι ώστε να μπορούμε να αλλάζουμε την τιμή αυτού του δείκτη, ακόμη και στην περίπτωση που είναι το πρώτο στοιχείο και πρέπει να αλλάξουμε το πεδίο **first** του κουβά, χρησιμοποιούμε έναν διπλό δείκτη **Node**.

Ο διπλός αυτός δείκτης ξεκινά από την θέση του δείκτη **first** του κουβά και συνεχίζει στα πεδία **next** των επόμενων κόμβων, μέχρι το επόμενο στοιχείο να έχει την ημερομηνία που θέλουμε ή να είναι **NULL**.

Εφόσον βρεθεί, στον δείκτη που δείχνει ο διπλός δείκτης ανατίθεται η τιμή του πεδίου **next** του κόμβου που διαγράφεται.

Στην περίπτωση που διαγράφουμε το τελευταίο στοιχείο του κουβά, ενημερώνεται κατάλληλα το πεδίο **last** του κουβά.

Κεφάλαιο 6

Το τελικό πρόγραμμα

Στο τελευταίο αυτό κεφάλαιο θα δούμε πώς συνθέτουμε όλα τα επιμέρους κομμάτια του Μέρους II σε ένα ενιαίο πρόγραμμα στο οποίο ο χρήστης μπορεί να περιηγηθεί μέσω ενός μενού. Αρχικά θα δούμε τη δομή του μενού, καθώς επίσης και τη λογική πίσω από τον σχεδιασμό του και στη συνέχεια θα εξετάσουμε κάποιες από τις λειτουργίες που μπορεί να εκτελέσει ο χρήστης πάνω στα δεδομένα.

Ενότητα 6.1 Συναρτήσεις του μενού

Πριν δούμε πώς κατασκευάζουμε το μενού, θα αναλύσουμε κάποιες βασικές συναρτήσεις που χρησιμοποιούμε στη συνέχεια:

6.1.1 Η συνάρτηση `menuOption`

Δήλωση:

```
void menuOption(int* option, int lower, int upper)
```

Παράμετροι:

- **option** – Δείκτης προς τη μεταβλητή στην οποία θα αποθηκευτεί η επιλεγμένη τιμή.
- **lower** – Κάτω φράγμα για τις επιτρεπτές επιλογές.
- **upper** – Άνω φράγμα για τις επιτρεπτές επιλογές

Η συνάρτηση `menuOption` δέχεται ως ορίσματα δύο ακεραίους **lower** και **upper** και επιτρέπει στον χρήστη να κάνει μια επιλογή ενός ακεραίου στο κλειστό διάστημα `[lower, upper]`. Χρησιμοποιείται στο μενού, ώστε ο χρήστης να μπορεί να επιλέξει μια λειτουργία από ένα σύνολο επιτρεπτών επιλογών.

6.1.2 Η συνάρτηση `inputDate`

Δήλωση:

```
Date inputDate(void)
```

Επιστρέφει:

Ένα struct τύπου `Date` με την ημερομηνία που εισήγαγε ο χρήστης.

Η συνάρτηση `inputDate` επιτρέπει στον χρήστη να εισάγει μια ημερομηνία της μορφής DD/MM/YYYY, με τους περιορισμούς ότι $DD \in [1, 31]$, $MM \in [1, 12]$ και $YYYY \in [2000, 2019]$. Η ημερομηνία που εισάχθηκε επιστρέφεται έπειτα από τη συνάρτηση.

Ενότητα 6.2 Σχεδίαση του μενού

Το μενού του προγράμματος σχεδιάστηκε ως μια σειρά από εμφωλευμένους βρόχους `do...while`, οι οποίοι τρέχουν όσο μια μεταβλητή επιλογής δεν έχει πάρει τιμή που δηλώνει πως ο χρήστης πρέπει να πάει σε προηγούμενο μενού. Για παράδειγμα, το αρχικό μενού έχει την παρακάτω δομή:

```
int mainMenuOpt;
do
{
    .
    .
    .
}
while (mainMenuOpt != 3);
```

όπου θεωρούμε ότι για `mainMenuOpt = 3` το πρόγραμμα λήγει.

Ο χρήστης κάνει μια επιλογή από ένα επιτρεπτό σύνολο τιμών μέσω της συνάρτησης `menuOption`. Για παράδειγμα, στο βασικό μενού ο χρήστης έχει τις παρακάτω επιλογές:

1. Φόρτωση αρχείου σε AVL δέντρο
2. Φόρτωση αρχείου σε δομή Hashing
3. Έξοδος από την εφαρμογή

άρα καλούμε τη συνάρτηση ως εξής:

```
menuOption(&mainMenuOpt, 1, 3);
```

Αφού ο χρήστης κάνει μια επιλογή μέσα στο επιτρεπτό διάστημα, χρησιμοποιούμε μια συνθήκη **switch** για να προσδιορίσουμε σε ποιο μενού πρέπει να μεταφερθεί ο χρήστης:

```
switch(mainMenuOpt)
{
    case 1: //AVL
    {
        .
        .
        .
    }
    break;

    case 2: //Hashing
    {
        .
        .
        .
    }
    break;
}
```

Αφού οι πρώτες δύο επιλογές οδηγούν σε ένα νέο μενού, κάθε ένα από αυτά τα **case** θα περιέχει ό,τι περιγράψαμε παραπάνω και για το βασικό μενού:

```
case 1: //AVL
{
    int avlOption;
    do
    {
        printf("\nChoose the type of AVL tree:\n\t1. Date-based\n\t2.
Temperature-based\n\t3. Go back\n");

        menuOption(&avlOption, 1, 3);

        switch(avlOption)
        {
            case 1: //AVL -> Date-based AVL
            {
                .
                .
                .
            }
            break;

            case 2: //AVL -> Temp-based AVL
            {
                .
                .
                .
            }
        }
    }
}
```

```

        break;
    }
}
while (avlOption != 3);
}
break;

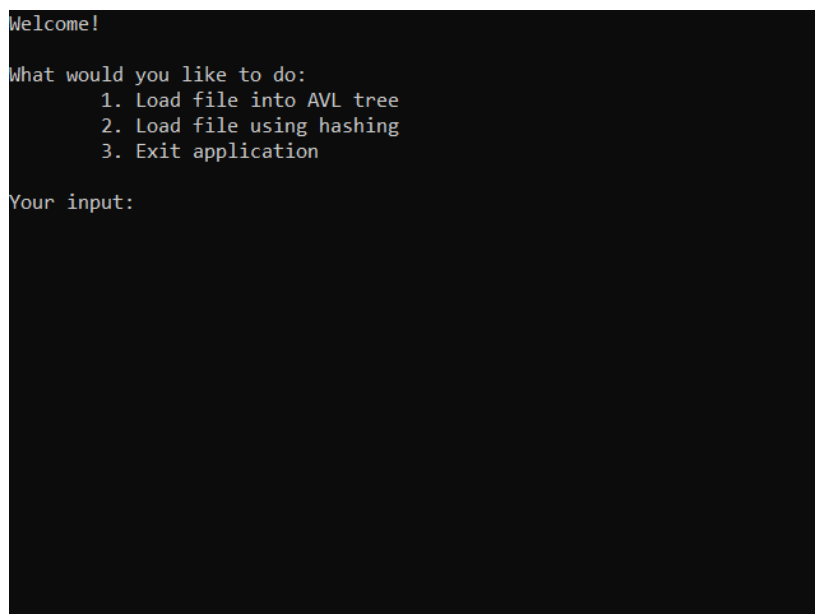
```

Αν ο χρήστης επιλέξει να πάει πίσω (στην παραπάνω περίπτωση για **avlOption** = 3), τότε βγαίνει από την επανάληψη αυτού του μενού και γυρνάει στην προηγούμενη (άρα και στο προηγούμενο μενού).

Η ίδια δομή επαναλαμβάνεται μέχρι τελικά να φτάσουμε σε τελικό μενού όπου μπορούμε να εκτελέσουμε κάποια λειτουργία. Τότε κάθε **case** απλά εκτελεί την περιγραφόμενη λειτουργία.

Ενότητα 6.3 Λειτουργίες

Τώρα που είδαμε πώς υλοποιείται το μενού, θα εξετάσουμε όλες τις λειτουργίες που μπορούμε να εκτελέσουμε στο τελικό πρόγραμμα. Τρέχοντας για πρώτη φορά την εφαρμογή, ο χρήστης συναντά την παρακάτω οθόνη:



```

Welcome!

What would you like to do:
    1. Load file into AVL tree
    2. Load file using hashing
    3. Exit application

Your input:

```

Εικόνα 6.1 Αρχικό μενού

όπου προτρέπεται να κάνει μια επιλογή. Αν ο χρήστης επιλέξει να φορτώσει το αρχείο σε δέντρο AVL, τότε μπορεί να επιλέξει αν θέλει το AVL να είναι διατεταγμένο ως προς τις ημερομηνίες ή τις θερμοκρασίες:

```
Welcome!

What would you like to do:
    1. Load file into AVL tree
    2. Load file using hashing
    3. Exit application

Your input: 1

Choose the type of AVL tree:
    1. Date-based
    2. Temperature-based
    3. Go back

Your input:
```

Εικόνα 6.2 Αρχικό μενού -> AVL

Έστω ότι επιλέγουμε AVL ημερομηνιών. Τότε:

```
Welcome!

What would you like to do:
    1. Load file into AVL tree
    2. Load file using hashing
    3. Exit application

Your input: 1

Choose the type of AVL tree:
    1. Date-based
    2. Temperature-based
    3. Go back

Your input: 1

Created AVL tree

What would you like to do:
    1. Print tree in-order
    2. Search temperatures for given date
    3. Modify temperatures for given date
    4. Delete node
    5. Go back
```

Εικόνα 6.3 Αρχικό μενού -> AVL -> AVL ημερομηνιών

Δημιουργείται ένα AVL ημερομηνιών από το αρχείο **ocean.csv**. Εδώ μπορούμε να εφαρμόσουμε διάφορες πράξεις πάνω στο δέντρο.

Η εκτύπωση του AVL με ενδοδιατεταγμένη διαπέραση εμφανίζει το παρακάτω αποτέλεσμα (προφανώς δεν είναι ολόκληρο το δέντρο):

```

=====
Date: 7/1/2000
Temperature: 11.000

Date: 7/1/2000
Temperature: 11.000

Date: 8/1/2000
Temperature: 11.210

Date: 8/1/2000
Temperature: 11.210

Date: 9/1/2000
Temperature: 17.950

Date: 9/1/2000
Temperature: 17.950

Date: 10/1/2000
Temperature: 17.910

Date: 10/1/2000
Temperature: 17.910

```

Εικόνα 6.4 Αρχικό μενού -> AVL -> AVL ημερομηνιών -> Ενδοδιατεταγμένη διαπέραση

Αν επιλέξουμε να αναζητήσουμε μια θερμοκρασία για δεδομένη μέρα, ζητείται από τον χρήστη να εισάγει μια ημερομηνία και στη συνέχεια γίνεται η εκτύπωση των θερμοκρασιών για τη μέρα εκείνη:

```

What would you like to do:
  1. Print tree in-order
  2. Search temperatures for given date
  3. Modify temperatures for given date
  4. Delete node
  5. Go back

Your input: 2

Please input a date:

Day: 1

Month: 2

Year: 2002

=====
Temperature: 14.730
=====

What would you like to do:
  1. Print tree in-order
  2. Search temperatures for given date

```

Εικόνα 6.5 Αρχικό μενού -> AVL -> AVL ημερομηνιών -> Αναζήτηση θερμοκρασιών για τη μέρα 1/2/2002

Αν επιλέξουμε να τροποποιήσουμε τις θερμοκρασίες για μια δεδομένη μέρα, ζητείται πάλι από τον χρήστη να εισάγει μια ημερομηνία. Επειδή οι θερμοκρασίες στο πρόγραμμά μας μπορεί να είναι παραπάνω από μια για μια ημέρα, αντί ο χρήστης να αλλάζει μόνο μια

επιλέξαμε να μπορεί να εισάγει ένα νέο διάνυσμα θερμοκρασιών. Επομένως στη συνέχεια ερωτάται πόσες θερμοκρασίες θέλει να έχει στο νέο διάνυσμα (οι παλιές θερμοκρασίες απορρίπτονται) και τελικά τις εισάγει:

```
Please input a date:
Day: 1
Month: 2
Year: 2002
How many temperatures do you want to insert? (between 1 and 12)
Your input: 5
Input temperature #1: 11
Input temperature #2: 12
Input temperature #3: 13
Input temperature #4: 14
Input temperature #5: 15
What would you like to do:
1. Print tree in-order
```

Εικόνα 6.6 Αρχικό μενού -> AVL -> AVL ημερομηνιών -> Τροποποίηση θερμοκρασιών της μέρας 1/2/2002

Με μια γρήγορη αναζήτηση μπορούμε να δούμε ότι όντως έγινε τροποποίηση των θερμοκρασιών:

```
What would you like to do:
1. Print tree in-order
2. Search temperatures for given date
3. Modify temperatures for given date
4. Delete node
5. Go back
Your input: 2
Please input a date:
Day: 1
Month: 2
Year: 2002
=====
Temperatures: 11.000, 12.000, 13.000, 14.000, 15.000
=====
What would you like to do:
1. Print tree in-order
2. Search temperatures for given date
```

Εικόνα 6.7 Αρχικό μενού -> AVL -> AVL ημερομηνιών -> Αναζήτηση θερμοκρασιών για τη μέρα 1/2/2002 μετά από τροποποίηση

Τέλος, μπορούμε να διαγράψουμε και έναν κόμβο, εισάγοντας την ημερομηνία του κόμβου που θέλουμε να διαγραφεί:

```
3. Modify temperatures for given date
4. Delete node
5. Go back

Your input: 4

Please input a date:

Day: 1

Month: 2

Year: 2002

Entry deleted successfully

What would you like to do:
1. Print tree in-order
2. Search temperatures for given date
3. Modify temperatures for given date
4. Delete node
5. Go back

Your input:
```

Εικόνα 6.8 Αρχικό μενού -> AVL -> AVL ημερομηνιών -> Διαγραφή της μέρας 1/2/2002

Ενημερωνόμαστε ότι η διαγραφή ήταν επιτυχής. Μπορούμε να το επιβεβαιώσουμε με μια αναζήτηση:

```
5. Go back

Your input: 2

Please input a date:

Day: 1

Month: 2

Year: 2002

=====
Could not find an entry with the given date
=====

What would you like to do:
1. Print tree in-order
2. Search temperatures for given date
3. Modify temperatures for given date
4. Delete node
5. Go back

Your input:
```

Εικόνα 6.9 Αρχικό μενού -> AVL -> AVL ημερομηνιών -> Αναζήτηση θερμοκρασιών για τη μέρα 1/2/2002 μετά από διαγραφή

Έστω τώρα ότι δεν θέλαμε AVL ημερομηνιών, αλλά AVL θερμοκρασιών. Τότε μπορούμε να πάμε πίσω και το AVL που δημιουργήσαμε διαγράφεται από τη μνήμη. Εκεί μπορούμε να επιλέξουμε την 2^η επιλογή, που είναι το AVL θερμοκρασιών:

```

    2. Search temperatures for given date
    3. Modify temperatures for given date
    4. Delete node
    5. Go back

Your input: 5

Deleted AVL tree from memory

Choose the type of AVL tree:
    1. Date-based
    2. Temperature-based
    3. Go back

Your input: 2

Created AVL tree

What would you like to do:
    1. Find days with minimum temperature
    2. Find days with maximum temperature
    3. Go back

Your input:

```

Εικόνα 6.10 Αρχικό μενού -> AVL -> AVL θερμοκρασιών

Εδώ έχουμε δύο επιλογές: είτε να βρούμε τις μέρες με την ελάχιστη θερμοκρασία, είτε αυτές με τη μέγιστη. Και οι δύο επιλογές φαίνονται παρακάτω:

```

Your input: 1

=====
Minimum temperature of 07.600 found in 1 dates:

30/4/2010
=====

What would you like to do:
    1. Find days with minimum temperature
    2. Find days with maximum temperature
    3. Go back

Your input: 2

=====
Maximum temperature of 25.420 found in 4 dates:

26/10/2018
25/10/2018
13/4/2019
10/4/2019
=====

```

Εικόνα 6.11 Αρχικό μενού -> AVL -> AVL θερμοκρασιών -> Μέγιστη και ελάχιστη θερμοκρασία

Έστω τώρα ότι δεν θέλαμε ούτε AVL θερμοκρασιών, αλλά δομή Hashing. Μπορούμε να πάμε πίσω δύο φορές στο μενού:

```
What would you like to do:
  1. Find days with minimum temperature
  2. Find days with maximum temperature
  3. Go back

Your input: 3

Deleted AVL tree from memory

Choose the type of AVL tree:
  1. Date-based
  2. Temperature-based
  3. Go back

Your input: 3

What would you like to do:
  1. Load file into AVL tree
  2. Load file using hashing
  3. Exit application

Your input:
```

Εικόνα 6.12 Πίσω στο αρχικό μενού

Όπου και επιλέγουμε τη δομή Hashing:

```
Welcome!

What would you like to do:
  1. Load file into AVL tree
  2. Load file using hashing
  3. Exit application

Your input: 2

Created hash table

What would you like to do:
  1. Search temperature
  2. Modify temperature
  3. Delete temperature
  4. Go back

Your input:
```

Εικόνα 6.13 Αρχικό μενού -> Hashing

Εδώ έχουμε τρεις επιλογές: αναζήτηση θερμοκρασίας, εύρεση μέγιστης θερμοκρασίας και διαγραφή θερμοκρασίας. Παρακάτω φαίνονται και οι τρεις λειτουργίες:

```
Please input a date:

Day: 26

Month: 3

Year: 2012

Search Found
Date: 2012-03-26, Temp: 15.64

What would you like to do:
    1. Search temperature
    2. Modify temperature
    3. Delete temperature
    4. Go back

Your input:
```

Εικόνα 6.14 Αρχικό μενού -> Hashing -> Αναζήτηση θερμοκρασίας για την ημερομηνία 26/3/2012

```
Please input a date:

Day: 26

Month: 3

Year: 2012

Search Found
Date: 2012-03-26, Temp: 15.64
Would you like to modify?
yes or no
yes

Enter a new temperature: 19.87

-New values-
Date: 2012-03-26, Temp: 19.87

What would you like to do:
    1. Search temperature
    2. Modify temperature
    3. Delete temperature
```

Εικόνα 6.15 Αρχικό μενού -> Hashing -> Τροποποίηση θερμοκρασίας της μέρας 26/3/2012

```
Your input: 3
Please input a date:
Day: 26
Month: 3
Year: 2012
Search Found
Date: 2012-03-26, Temp: 19.87
Would you like to delete?
yes or no
yes

What would you like to do:
    1. Search temperature
    2. Modify temperature
    3. Delete temperature
    4. Go back
Your input:
```

Εικόνα 6.16 Αρχικό μενού -> Hashing -> Διαγραφή της μέρας 26/3/2012

Github repository:

https://github.com/Nikolis2002/Project_DataStruct.git

