PROJECT: "DATA STRUCTURES 2023"

Από:
Κατσαντά Θοδωρή με ΑΜ 1097459
Δούβρη Αγγελική με ΑΜ 1097441

ΠΕΡΙΕΧΟΜΕΝΑ

Γενικός Σχεδιασμός των Προγραμμάτων

Στο zip αρχείο παρουσιάζεται το project στο μάθημα των δομών δεδομένων. Ο κώδικας αυτός καλύπτει και τα 2 part της εργασίας, και τρέχει όλος.

Παρακάτω θα αναφερθούν τα κομμάτια του πηγαίου κώδικα που θα χρησιμοποιηθούν σε όλο το Part1.

```c
#ifndef DATAENTRY_H
#define DATAENTRY_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


typedef struct DataEntry {
    char Direction[20];
    int Year;
    char Date[20];
    char Weekday[20];
    char Country[20];
    char Commodity[20];
    char Transport_mode[20];
    char Measure[3];
    int Value;
    unsigned int Cumulative;
} DataEntry;


#endif
```

Αρχικά έπρεπε το πρόγραμμα να αναγνωρίζει το αρχείο με τα Effects of COVID-19 on trade. Οπότε φτιάξαμε το **header file** με το όνομα **dataentry.h** ώστε να σκιαγραφήσουμε τον πίνακα που θα επεξεργαστούμε αργότερα. Δημιουργήσαμε μια δομή με όνομα DataEntry χρησιμοποιώντας το «typedef struct», που περιέχει με char τις στήλες Direction, Date, Weekday, Country, Commodity, Transport_mode και Measure και με int Year, Value και Cumulative. Έτσι φτιάχνουμε τον ίδιο πίνακα με το αρχείο που θέλουμε να ταξινομήσουμε. Και με το #ifndef και το #endif στην αρχή και στο τέλος του φακέλου αντίστοιχα δίνεται ο ορισμός της DATAENTRY για να συμπεριληφθεί αυτός ο κώδικας στο πρόγραμμά μας.

Τώρα θα αναφερθούμε στην main μας. Στην αρχή κάνουμε #include τον φάκελο dataentry.h και καθορίζουμε INITIAL_CAPACITY 1000 και CAPACITY_INCREMENT_PERCENT 50 γιατί ιδανικά θέλουμε να έχουμε την μικρότερη δυνατή δέσμευση μνήμης. Σε περίπτωση που ο πίνακας του αρχείου csv έχει περισσότερες γραμμές από αυτές που δώσαμε σαν δεδομένες, αμέσως μετά χρησιμοποιούμε την **resizeEntries**. Όταν καθώς περνάνε τα

```c
void resizeEntries (DataEntry **entries , int *capacity) {
    int newCapacity = (*capacity) + ((*capacity) * CAPACITY_INCREMENT_PERCENT) / 100;
    DataEntry *newEntries = realloc (*entries, newCapacity * sizeof(DataEntry));
    if (newEntries == NULL) {
        printf("Memory reallocation failed for entries array\n");
        // Handle the error
    } else {
        *entries = newEntries;
        *capacity = newCapacity;
    }
}
```

δεδομένα στον πίνακά μας καλυφθεί όλη η μνήμη, καλείται η συγκεκριμένη συνάρτηση και δεσμεύει περισσότερη. Και συνεχίζει έτσι κάθε φορά που γεμίζει ο

```
int countLines (FILE *file) {
    int count = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        if (ch == '\n') {
            count++;
        }
    }
    rewind (file);
    return count;
}
```

πίνακας, μέχρι να περάσουν όλα τα στοιχεία. Αφού περαστούν τα δεδομένα, θα πρέπει το πρόγραμμα να διαβάζει το αρχείο, οπότε δημιουργήσαμε την συνάρτηση **countLines**.

Αφού ολοκληρώσαμε τις απαραίτητες προεργασίες, εστιάζουμε στην main. Στην αρχή δίνουμε την εντολή στο πρόγραμμα να διαβάσει το αρχείο με τα Effects of COVID-19 on trade που το ονομάσαμε effects.csv για δικιά μας ευκολία. Σε περίπτωση που για κάποιον λόγο δεν μπορεί να ανοιχτεί το αρχείο, το πρόγραμμα εκτυπώνει στην οθόνη ‘’Error opening effects.csv’’. Αφού ανοιχτεί το αρχείο, θέλουμε να αποθηκευτούν κατευθείαν τα στοιχεία στον πίνακά μας και όχι οι αντίστοιχες επικεφαλίδες. Για αυτό ορίζουμε το **headers**. Σε περίπτωση που δεν αγνοήσει τις επικεφαλίδες, θα εκτυπώνει το μήνυμα ‘’ Error reading headers’’. Αμέσως μετά δεσμεύουμε δυναμικά μνήμη για την δομή με την χρήση της malloc. Αν αποτύχει αυτή η ενέργεια, εμφανίζει το αντίστοιχο μήνυμα "Memory allocation failed for entries array".

```
FILE *file = fopen("effects.csv", "r");
if (file == NULL) {
    printf("Error opening effects.csv\n");
    return 1;
}
```

```
char headers[256];
if (fgets(headers, sizeof(headers), file) == NULL) {
    printf("Error reading headers\n");
    fclose(file);
    return 1;
}
```

```
char line[256];
int entryCount = 0;
int entryCapacity = INITIAL_CAPACITY;
DataEntry *entries = malloc(entryCapacity * sizeof(DataEntry));
if (entries == NULL) {
    printf("Memory allocation failed for entries array\n");
    fclose(file);
    return 1;
}
```

Ύστερα, καθώς το πρόγραμμα διαβάζει το csv, αντιμετωπίσαμε μια δυσκολία στην αναγνώριση ορισμένων χαρακτήρων όπως τον «$». Δηλαδή το πρόγραμμα εκτύπωνε όλες τις στήλες, μέχρι να φτάσει στην 8η, το measure. Οπότε προσθέσαμε

```
while (fgets(line, sizeof(line), file)) {
    DataEntry entry;
    int result;
    if (strchr(line, '\"') != NULL) {
        result = sscanf(line, " %[^,],%d,%[^,],%[^,],%[^,],\"%[^\"]\",%[^,],%[^,],%d,%u",
                        entry.Direction, &entry.Year, entry.Date, entry.Weekday,
                        entry.Country, entry.Commodity, entry.Transport_mode,
                        entry.Measure, &entry.Value, &entry.Cumulative);
    } else {
        result = sscanf(line, " %[^,],%d,%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%d,%u",
                        entry.Direction, &entry.Year, entry.Date, entry.Weekday,
                        entry.Country, entry.Commodity, entry.Transport_mode,
                        entry.Measure, &entry.Value, &entry.Cumulative);
    }

    if (result != 10) {
        printf("Error reading data from line: %s\n", line);
        fclose(file);
        free(entries);
        return 1;
    }
```

τον συγκεκριμένο κώδικα για την αντιμετώπιση αυτού του θέματος. Άμα προκύψει και άλλο θέμα στο ‘’διάβασμα’’ , το πρόγραμμα θα εκτυπώσει «Error reading data from line: %s» και παρουσιάζει σε ποια γραμμή έχει εμφανιστεί θέμα.

Συνεχίζοντας, άμα γεμίσει η μνήμη που έχουμε δεσμεύσει, καλούμε την **resizeEntries**.

Και με αυτό τον τρόπο έχουμε τον πίνακά μας με τα δεδομένα του.

Part 1

1)Counting & Merge Sort

Μπαίνοντας στο **part 1,** δίνουμε στον χρήστη στην main την δυνατότητα να επιλέξει πως θέλει να γίνει η ταξινόμηση, είτε με 1.Counting Sort είτε με 2.Merge Sort. Άμα θελήσει να κλείσει το πρόγραμμα επιλέγει την 3.Terminate. Οπότε συμπληρώνει ο χρήστης 1,2 ή 3. Άμα δεν γράψουν έναν από αυτούς τους αριθμούς θα εμφανιστεί το μήνυμα «Invalid choice. Please try again.». Έχουμε φτιάξει διαφορετικά c files για την counting sort και την merge sort που θα εξηγήσουμε αναλυτικά πιο κάτω.

Στο τέλος, αφού γίνει η ταξινόμηση, προσθέσαμε την συγκεκριμένη σειρά εντολών που εμφανίζει όλα τα αποτελέσματα σε txt file, επειδή προηγουμένως δεν εκτύπωνε όλο τον πίνακα ταξινομημένο. Ύστερα δίνουμε τις αντίστοιχες εντολές για την εκτύπωση του πίνακα με αύξουσα σειρά και αμέσως μετά κάνουμε έλεγχο του πίνακα με την χρήση if statement για να δούμε αν είναι σωστό το αποτέλεσμα, και σε περίπτωση που δεν είναι, εκτυπώνει το πρόγραμμα μας στην οθόνη το αντίστοιχο μήνυμα. Έτσι δημιουργούμε μια δικλείδα ασφαλείας ώστε να διασφαλίσουμε την αποτελεσματικότητα και των 2 μεθόδων.

```
FILE *outputFile = fopen("sorted_entries.txt", "w");
if (outputFile == NULL) {
    printf("Error opening output file\n");
    free(entries);
    return 1;
}
```

```
switch (choice) {
    case 1:
        CountingSort(entries, entryCount, maxValue);
        printf("Counting Sort has been applied.\n");
        break;
    case 2:
        MergeSort(entries, 0, entryCount - 1);
        printf("Merge Sort has been applied.\n");
        break;
    case 3:
        printf("Terminating the program.\n");
        free(entries);
        return 0;
    default:
        printf("Invalid choice. Please try again.\n");
`
```

```
int error = 0;
for (int i = 1; i < entryCount; i++) {
    if (entries[i].Value < entries[i - 1].Value) {
        fprintf(outputFile, "Error: Entries are not sorted correctly\n");
        error++;
    }
}

fprintf(outputFile, "Errors: %d\n", error);
```

Counting Sort

Αρχικά για το Counting Sort εστιάζουμε στο αντίστοιχο file με όνομα countingsort.c. Δημιουργούμε αυτή την συνάρτηση που παίρνει 3 ορίσματα, το arr που είναι ο πίνακας από το dataentry, το n που είναι ο αριθμός των στοιχείων μέσα στον πίνακα και το maxValue που είναι ο μέγιστος αριθμός στοιχείων στον πίνακα. Ύστερα δεσμεύουμε μνήμη για έναν πίνακα count

```
void CountingSort(DataEntry* arr, int n, int maxValue)
{
    int* count = malloc((maxValue + 1) * sizeof(int));
    if (count == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    for (int i = 0; i <= maxValue; i++)
    {
        count[i] = 0;
    }
```

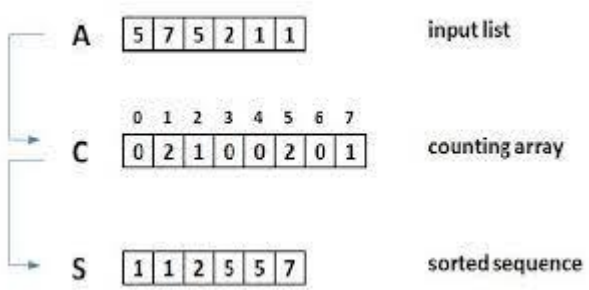
(μεγέθους ίσου με το μέγιστο στοιχείο) με την χρήση της malloc και τον αρχικοποιούμε με όλα τα στοιχεία ίσα με το 0.

Παίρνουμε τον αρχικό πίνακα και τον διατρέχουμε αυξάνοντας το αντίστοιχο στοιχείο του count πίνακα για κάθε τιμή που συναντάμε. Αναπροσαρμόζουμε συνεχώς τον ίδιο πίνακα έτσι ώστε κάθε στοιχείο να αντιπροσωπεύει τον αριθμό των στοιχείων που είναι μικρότεροι ή ίσοι από την τιμή που βρισκόμαστε. Δημιουργούμε και έναν προσωρινό πίνακα sortedArray για να αποθηκευτούν τα ταξινομημένα στοιχεία. Από το τέλος του αρχικού πίνακα, τοποθετούμε τα στοιχεία στην σωστή θέση στον προσωρινό πίνακα, βασιζόμενοι στις μετρήσεις που έχουμε καταγράψει στον "count" πίνακα, δηλαδή πόσες τιμές υπάρχουν πολλαπλές φορές.

```
for (int i = 1; i <= maxValue; i++)
{
    count[i] += count[i - 1];
}
DataEntry* sortedArray = malloc(n * sizeof(DataEntry));
if (sortedArray == NULL)
{
    printf("Memory allocation failed\n");
    free(count);
    return;
}
for (int i = n - 1; i >= 0; i--)
{
    int value = arr[i].Value;
    int index = count[value] - 1;
    sortedArray[index] = arr[i];
    count[value]--;
}
memcpy(arr, sortedArray, n * sizeof(DataEntry));
```

```
free(sortedArray);
free(count);
```

Και τέλος αντιγράφουμε τα ταξινομημένα στοιχεία από τον προσωρινό πίνακα πίσω στον αρχικό πίνακα και απελευθερώνουμε την μνήμη. Πιο συγκεκριμένα, η διαδικασία που ακολουθήσαμε παρουσιάζεται σε αυτή την εικόνα:

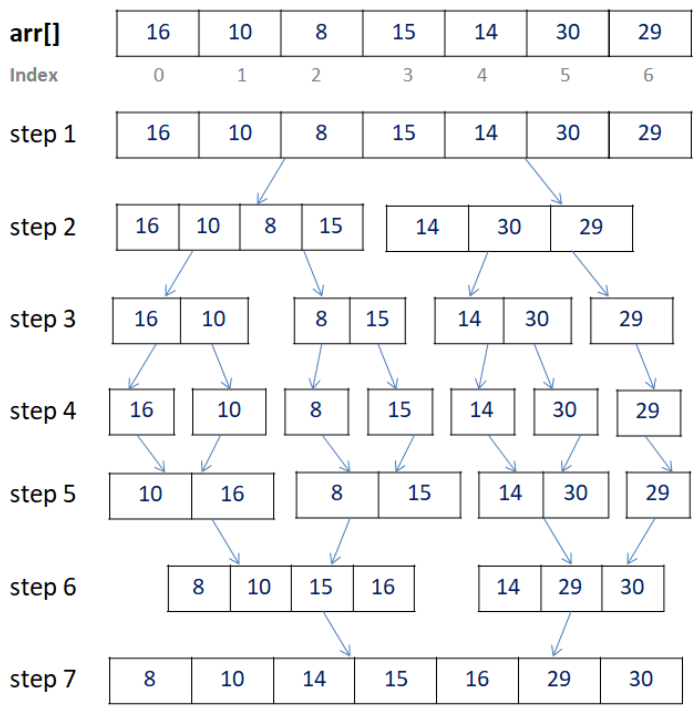


Merge Sort

```
void Merge(DataEntry *arr, int low, int mid, int high) {
    int leftSize = mid - low + 1;
    int rightSize = high - mid;

    DataEntry *left = malloc(leftSize *
    DataEntry *right = malloc(rightSize

    for (int i = 0; i < leftSize; i++)
        left[i] = arr[low + i];
    }
    for (int j = 0; j < rightSize; j++) }
        right[j] = arr[mid + 1 + j];
int i = 0; // Deikths gia ton aristero pinaka
int j = 0; // Deikths gia ton dexio pinaka
int k = low; // Deikths gia ton enwmeno pinaka
while (i < leftSize && j < rightSize) {
    if (left[i].Value <= right[j].Value) {
        arr[k] = left[i];
        i++;
    } else {
        arr[k] = right[j];
        j++;
    }
    k++;
}
while (i < leftSize) {
    arr[k] = left[i];
    i++;
    k++;
}
while (j < rightSize) {
    arr[k] = right[j];
    j++;
    k++;
}
```

```
void MergeSort(DataEntry *arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        MergeSort(arr, low, mid);
        MergeSort(arr, mid + 1, high);
        Merge(arr, low, mid, high);
    }
```

Για το Merge sort εστιάζουμε στον φάκελο mergesort.c. Αρχικά δημιουργούμε προσωρινούς πίνακες για αριστερά κα δεξιά subarrays με τον αριστερό να περιέχει τα μισά δεδομένα και τον δεξιό τα υπόλοιπα. Μετά αντιγράφουμε τα δεδομένα μας σε αυτούς τους πίνακες. Η **Merge** διαιρεί κάθε φορά τους πίνακες στην μέση σε όλο και μικρότερους μέχρι να περιέχει κάθε πίνακας ένα μόνο στοιχείο. Ύστερα συγκρίνοντας τους προσωρινούς υποπίνακες με το ένα στοιχείο σταδιακά τους συγχωνεύουμε χρησιμοποιώντας 3 loop με while, μεταφέροντας όλα τα στοιχεία στον πίνακά μας ταξινομημένα. Στο τέλος ελευθερώνουμε και την μνήμη (τους υποπίνακες δηλαδή).



Ο αρχικός πίνακας

Χώρισε τον πίνακα σε 2 υποπίνακες

Επανάλαβε αυτή την διαδικασία μέχρι σε κάθε πίνακα να έχεις 2 στοιχεία

"Σπάσε" ξανά τους πίνακες μέχρι ο καθένας να έχει 1 στοιχείο

Ταξινόμησε τα νούμερα από το μικρό στο μεγαλύτερο

Συγχώνευσε τους πίνακες διατηρώντας την αύξουσα σειρά

Ο πίνακας έχει ταξινομηθεί

Η merge sort μπορούμε να αντιληφθούμε καλύτερα πως λειτουργεί με την βοήθεια αυτής της εικόνας.

Σε αυτό το σημείο μπορούμε να συγκρίνουμε πειραματικά αυτές τις δύο μεθόδους ταξινόμησης. Ο αλγόριθμος Counting Sort είναι αποδοτικός όταν η τιμή εύρους των στοιχείων είναι μικρή σε σχέση με τον αριθμό των στοιχείων που πρέπει να ταξινομηθούν. Ωστόσο, απαιτεί μεγαλύτερη δέσμευση μνήμης για τον προσωρινό πίνακα sortedArray και τον πίνακα count και όταν πρόκειται για μεγάλους πίνακες , θα απαιτήσει και περισσότερο χρόνο καθώς έχει γραμμική πολυπλοκότητα χρόνου (O(n+k) όπου n ο αριθμός στοιχείων και k η μέγιστη τιμή του πίνακα).
`execution time : 4.062 s` *Αντιθέτως, ο Merge Sort είναι καλύτερος στους μεγάλους πίνακες με πολλά στοιχεία καθώς τον ταξινομεί με μεγαλύτερη ταχύτητα αφού έχει χειρότερη πολυπλοκότητα χρόνου(O(nlogn) όπου n ο αριθμός στοιχείων)*
`execution time : 0.856 s` *αλλά και δεν χρειάζεται επιπλέον μνήμη εκτός του αρχικού πίνακα και των προσωρινών υποπινάκων.*

2)Heap & Quick Sort

Η διαφοροποίηση που έχει γίνει εδώ στην main είναι τα παρακάτω:

1.οι επιλογές που δίνονται στον χρήστη περί ποιανής μεθόδου ταξινόμησης θα χρησιμοποιήσει είναι 1. Heap Sort, 2. Quick Sort και 3. Terminate.

```
case 1:
    heapSort(entries, entryCount);
    printf("Heap Sort has been applied.\n");
    break;
case 2:
    quickSort(entries, 0, entryCount - 1);
    printf("Quick Sort has been applied.\n");
    break;
case 3:
    printf("Terminating the program.\n");
    free(entries);
    return 0;
default:
    printf("Invalid choice. Please try again.\n");
    free(entries);
    return 1;}
```

2.στο switch (choice) τώρα πια καλούνται οι συναρτήσεις που επέλεξε ο δέκτης και εκτυπώνουν στην οθόνη το αντίστοιχο μήνυμα, ότι δηλαδή η επιλογή που έκανε είναι πράγματι το sort που επιλέξανε.

3.Και τέλος, έχουμε βάλει έναν έλεγχο ,ότι η επιλογή που έχει γίνει είναι σωστή.

Εκτός από την main , προσθέσαμε εδώ και άλλο ένα header file εκτός από την dataentry με όνομα sorting.h. Σε αυτόν το φάκελο ορίζουμε για την Heap Sort, functions όπως την heapify που θα διαμορφώσει τον πίνακα arr σε μορφή heap και την heapSort που θα υλοποιεί την Heap Sort καλώντας την heapify που θα τροποποιεί τον πίνακα και ύστερα θα μεταφέρει τον μεγαλύτερο αριθμό στο τέλος του πίνακα. Ταυτόχρονα ορίζουμε functions και για την Quick Sort που είναι η 1.partition (που θα χρησιμοποιεί το pivot για την οργάνωση του πίνακα) και η

2. quickSort που υλοποιεί τον αλγόριθμο Quick Sort με το partition

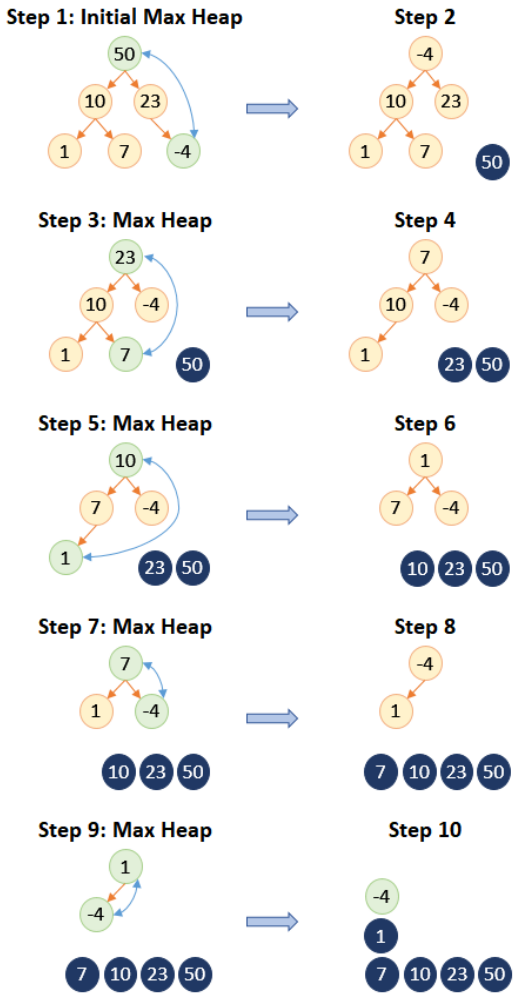
Και στο τέλος ορίζουμε στην βιβλιοθήκη και την swap που θα χρησιμοποιείται από τις παραπάνω.

2)Heap Sort

```
// arr: pointer ston pinaka
//n: ari8mos stoixeiwn
//i: deikths tou trexontos stoixeiou
void heapify(DataEntry *arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left].Cumulative > arr[largest].Cumulative) {
    largest = left;
}
    if (right < n && arr[right].Cumulative > arr[largest].Cumulative) {
    largest = right;
}

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}
```

Στο heapsort.c έχουμε διαμορφώσει τον κώδικα κατάλληλα ώστε να εκτελεί την ταξινόμηση αυτή. Πιο αναλυτικά, δημιουργούμε έναν φάκελο με όνομα heapsort.c και μέσα φτιάχνουμε με την **heapify** έναν πίνακα μορφής heap. Αυτή η συνάρτηση έχει ως είσοδο έναν πίνακα arr με μέγεθος έναν αριθμό n και τον δείκτη i (το οποίο θα χρησιμοποιηθεί σαν προσωρινή θέση για το στοιχείο που θα εξετάζουμε κάθε στιγμή).Η συγκεκριμένη συνάρτηση θα βλέπει το i και θα το συγκρίνει με τα παιδιά του(left και right), και σε περίπτωση που είναι μικρότερα από αυτό, θα αλλάζουν θέση. Ύστερα θα εκτελέσει πάλι την ίδια διαδικασία για το παιδί που άλλαξε. Αμέσως μετά δημιουργούμε άλλη μία συνάρτηση με το όνομα **heapSort** η οποία θα καλεί την **heapify** σε κάθε κόμβο και θα αλλάζει την σειρά των δεδομένων μέχρι να φτάσουμε σε ένα ταξινομημένο δέντρο. Από την ρίζα του παίρνουμε το μεγαλύτερο νούμερο και το τοποθετούμε στον πίνακα, και

```
void heapSort(DataEntry *arr, int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
```



επαναλαμβάνουμε (την ταξινόμηση και την τοποθέτηση) μέχρι να τοποθετηθούν όλα τα δεδομένα.

Με έναν διαφορετικό τρόπο, στην παρακάτω εικόνα απεικονίζεται ο heap sort.

Όπως φαίνεται και στο σχήμα, σε κάθε βήμα υπάρχει είτε μια σύγκριση μεταξύ γονέα και παιδιού είτε ένα swap μεταξύ αυτών των 2 και τοποθέτησή του μεγαλύτερου στον πίνακα. Κάθε φορά που αποσπάται ένα στοιχείο από το δέντρο, μεταφέρεται στον πίνακα, από δεξιά στα αριστερά. Μέχρι να τοποθετηθούν όλα τα στοιχεία στον πίνακα, οποίος είναι τώρα ταξινομημένος.

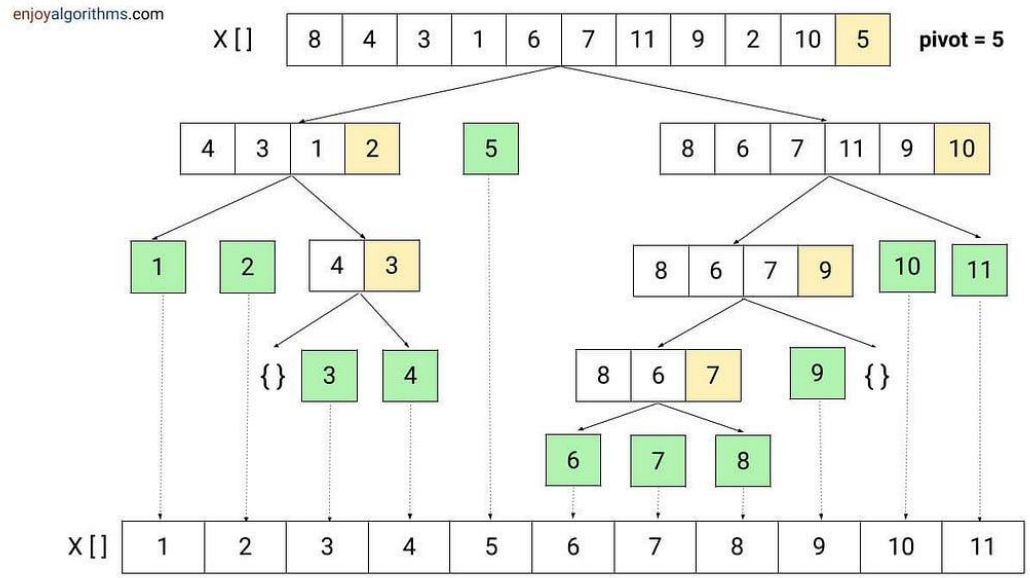
2)Quick Sort

Στην quicksort.c ορίζουμε πολύ απλά την swap, κα συνεχίζουμε τον ορισμό της partition και του pivot και τροποποιούμε τον κώδικα ώστε να εκτελείται αποτελεσματικά αυτή η μέθοδος ταξινόμησης, και έχουμε σαν ορίσματα τον πίνακα arr,την αριστερή μεριά του πίνακα low και την δεξιά μεριά high. Διαλέγουμε το στοιχείο που βρίσκεται τέρμα δεξιά στον πίνακα για pivot, και συγκρίνουμε τα υπόλοιπα στοιχεία με αυτό. Αμέσως μετά, τοποθετούμε όλα τα στοιχεία που ήταν μικρότερα

```
int partition(DataEntry *arr, int low, int high) {
    DataEntry pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j].Cumulative < pivot.Cumulative) {
    i++;
    swap(&arr[i], &arr[j]);
}

    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
```

από το pivot σε έναν πίνακα, και αυτά που ήταν μεγαλύτερα σε άλλον. Σε περίπτωση που βρεθεί στοιχείο με την ίδια τιμή με το pivot, θα μπει στο πίνακα που είναι στα δεξιά του πίνακα, δηλαδή στον πίνακα με τα μεγαλύτερά του. Θα ξαναθέσουμε το τελευταίο στοιχείο στους καινούργιους πίνακες ως pivot και θα ξανακολουθήσουμε την ίδια διαδικασία, μέχρι κάθε πίνακας να έχει μόνο ένα στοιχείο. Εκείνη την στιγμή ο πίνακας θα είναι ταξινομημένος.

```
void quickSort(DataEntry *arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```



Πιο κάτω θα συγκρίνουμε τις 2 αυτές μεθόδους ταξινόμησης. Η πολυπλοκότητα χρόνου ταξινόμησης τους είναι O(nlogn). Η heap sort έχει πιο σταθερή αποδοτικότητα από την quick sort(γιατί η heapify ελέγχει κάθε στοιχείο και τα συγκρίνει με τους γονείς του, οπότε επιβεβαιώνει ότι αυτοί είναι μεγαλύτεροι από αυτό), αλλά και δεσμεύει επιπλέον μνήμη για την δημιουργία του heap. Αντιθέτως η quick sort δεν χρειάζεται επιπλέον μνήμη για την εκτέλεσή της. Επιπλέον, αν ο πίνακας είναι ήδη ταξινομημένος, ή χρειάζεται λίγες αλλαγές , η heap sort θα το ελέγξει και θα τον παρουσιάσει ταξινομημένο με πολύ μεγαλύτερη ταχύτητα από την quick sort, η οποία θα απαιτήσει περισσότερο χρόνο από τον αναμενόμενο για να την παρουσιάσει. Συνεπώς, αν αγνοηθεί αυτή η περίπτωση (ότι δηλαδή ο πίνακας είναι ήδη ταξινομημένος), η quick sort είναι περισσότερο αποτελεσματική για μεγάλους και αταξινόμητους πίνακες ωστόσο η heap sort είναι πιο σταθερή αλλά δεσμεύει περισσότερη μνήμη από την άλλη μέθοδο ταξινόμησης.

3)Binary & Interpolation Search

Για την άσκηση 3 κάνουμε μερικές αλλαγές στον κύριο κώδικα. Εστιάζοντας σε αυτόν, οι αλγόριθμοι απαιτούν έναν ταξινομημένο πίνακα. Οπότε από τη προηγούμενη άσκηση (στην 2) ενσωματώνουμε όλο τον κώδικα της quick sort. Ύστερα, στην main μας αλλάζουμε τις επιλογές κατάλληλα ώστε να διαλέγει ο χρήστης τον αλγόριθμο αναζήτησης που επιθυμεί να χρησιμοποιήσει(1. Binary Search ή 2. Interpolation Search). Σε περίπτωση που δεν πατηθεί ένα νούμερο μεταξύ του 1 και του 2, το πρόγραμμα εμφανίζει ένα κατάλληλο μήνυμα για να ενημερώσει ότι η επιλογή που έγινε δεν είναι κατάλληλη, κι παρουσιάζει ξανά την ίδια επιλογή. Αμέσως μετά ζητείται από το πρόγραμμα να συμπληρωθεί η ημερομηνία που επιθυμούμε να βρούμε τα value και τα cumulative που αποθηκεύεται στο string searchDate. Μετά ο κώδικας απαιτεί τουλάχιστον 45 δευτερόλεπτα για να ταξινομήσει τον πίνακά μας. Ύστερα χρειάζεται να διαλέξουμε αν θέλουμε τα value ή τα cumulative ή και τα δύο.

```
if (searchAlgorithm != 1 && searchAlgorithm != 2) {
    printf("Invalid search algorithm\n");
    continue; }
```

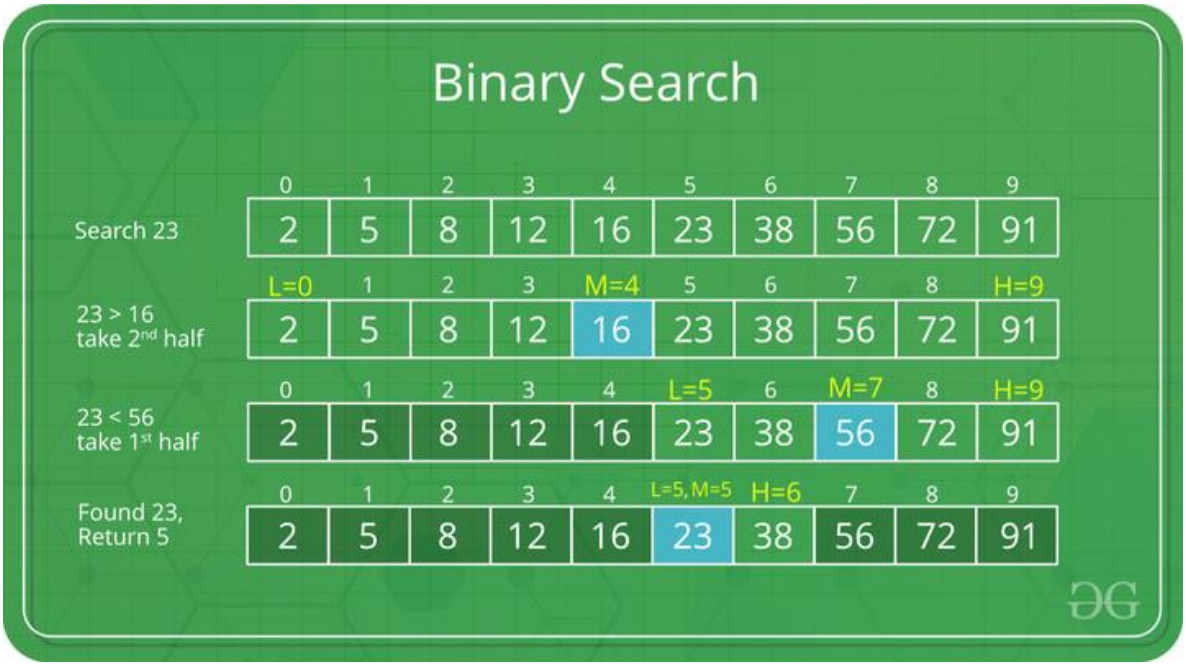
```
int compareDates(const char *date1, const char *date2) {
    int day1, month1, year1;
    int day2, month2, year2;
    sscanf(date1, "%d/%d/%d", &day1, &month1, &year1);
    sscanf(date2, "%d/%d/%d", &day2, &month2, &year2);

    if (year1 < year2) {
        return -1;
    } else if (year1 > year2) {
        return 1;
    } else {
        if (month1 < month2) {
            return -1;
        } else if (month1 > month2) {
            return 1;
        } else {
            if (day1 < day2) {
                return -1;
            } else if (day1 > day2) {
                return 1;
            } else {
                return 0;
            }
        }
    }
}
```

Σε αυτό το στάδιο θα φτιάξουμε μια συνάρτηση compareDates που θα είναι σε θέση να μπορεί να συγκρίνει ημερομηνίες. Αφού πρώτα μετατρέψει τις ημερομηνίες σε μορφή YYYYMMDD συγκρίνει τον χρόνο, μετά τον μήνα και μετά τις μέρες και να βρίσκει ποια ημερομηνία είναι μεγαλύτερη.

Πίσω στην main μας, το πρόγραμμα συγκρίνει σε όλο τον πίνακα τις ημερομηνίες, χρησιμοποιώντας τον αλγόριθμο που έχει επιλεγεί και κάθε φορά που μια ημερομηνία είναι ίση με αυτή που ψάχνουμε, εκτυπώνει στην οθόνη τα value και τα cumulative.

3)Binary Search

Ο Αλγόριθμος της δυαδικής αναζήτησης θα παρουσιαστεί πιο κάτω. Αφού ο χρήστης συμπληρώσει πρώτα ποιο νούμερο θέλει να βρεθεί, ο κώδικας παίρνει τον ταξινομημένο πίνακα και πάει στο στοιχείο που βρίσκεται στην μέση του. Αφού το βρεί, θα συγκρίνει το νούμερό μας με το στοιχείο. Αν είναι ίσα, τότε το βρήκαμε. Σε περίπτωση που είναι μικρότερο ή μεγαλύτερο, ο κώδικας θα καταλάβει ότι το νούμερο που αναζητείται δεν θα βρίσκεται από την δεξιά μεριά του πίνακα ή στην αριστερή αντίστοιχα. Οπότε αποκλείονται τα νούμερα που είναι από εκείνες τις μεριές. Ύστερα ακολουθείται η ίδια διαδικασία, δηλαδή διαιρείται ο υπόλοιπος πίνακας και συγκρίνεται ο αριθμός μας με το στοιχείο που ενώνει τα δύο κομμάτια, μέχρι να βρεθεί. Η εικόνα πιο κάτω παρουσιάζει ένα παράδειγμα αυτού του αλγορίθμου καθώς ψάχνει το νούμερο 23.



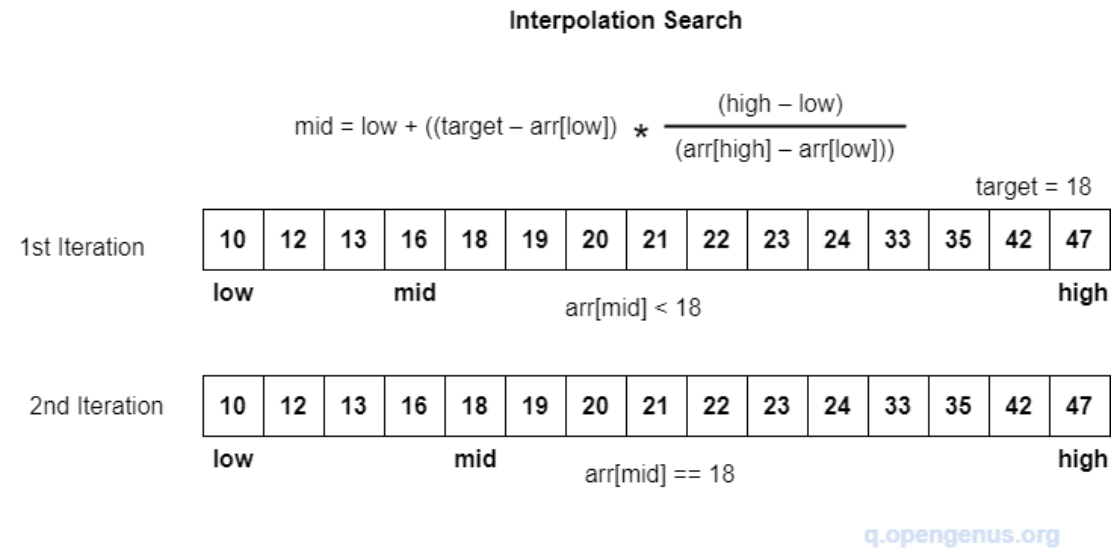
Στον κώδικά μας, για την υλοποίηση αυτού του τρόπου αναζήτησης δημιούργησε μια συνάρτηση με το όνομα binarySearch. Στην αρχή ορίζονται τα δύο άκρα του πίνακα (low and high) και εντοπίζεται η μέση του πίνακα (mid).Αμέσως μετά συγκρίνεται (με το if statement) η ημερομηνία που συμπλήρωσε ο χρήστης με την

```
int binarySearch(DataEntry *entries, int low, int high, const char *searchDate) {
    int foundIndex = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int comparison = compareDates(entries[mid].Date, searchDate);
        if (comparison == 0) {
            foundIndex = mid;
            high = mid - 1;
        } else if (comparison < 0) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return foundIndex;
}
```

mid. Αν είναι ίσα, ο δείκτης που είχαμε (comparison) γίνεται ίσος με 0 κα τελειώνει η έρευνα. Αν παραμείνει ο δείκτης αρνητικός, τότε η ημερομηνία που αναζητούμε είναι στα δεξιά του πίνακα, οπότε αλλάζουμε την τιμή που ήταν στο low με την τιμή του mid+1. Αν ο δείκτης γίνει θετικός τότε γίνεται η αντίθετη διαδικασία, και το high παίρνει την τιμή του mid-1. Η διαδικασία επαναλαμβάνεται με τον foundIndex να ενημερώνεται με την θέση mid μέχρι το comparison να πάρει την τιμή 0. Αν στο τέλος foundIndex=-1 σημαίνει ότι η ημερομηνία δεν βρέθηκε.

3)Interpolation Search

Η διαδοχική διαίρεση είναι ένας τρόπος αναζήτησης παρόμοιος με την δυαδική αναζήτηση. Στην αρχή, αντί να διαιρέσει τον πίνακα στην μέση, και να συγκρίνει την ημερομηνία που ψάχνουμε με την τιμή mid χρησιμοποιεί γραμμική παρεμβολή, έναν συγκεκριμένο τύπο που βασίζεται στις τιμές των άκρων και το αποτέλεσμα (position) της οποίας συγκρίνεται με αυτή. Σε περίπτωση που η ημερομηνία είναι μεγαλύτερη από το position, η αριστερή μεριά του πίνακα αποκλείεται, ενώ αν είναι μικρότερη, αποκλείεται η δεξιά μεριά. Αν είναι ίσες, τότε βρήκαμε την ημερομηνία. Ύστερα επαναχρησιμοποιούμε τον τύπο και με το αποτέλεσμα επαναλαμβάνουμε την ίδια διαδικασία μέχρι να βρούμε τη τιμή που αναζητούμε. Γενικά αυτός ο τρόπος αναζήτησης είναι πιο πρακτικός γιατί μετά από κάθε σύγκριση είναι δυνατόν να απορριφθεί ένα μεγάλο μέρος του πίνακα. Η εικόνα δίνει ένα παράδειγμα με έναν τυχαίο ταξινομημένο πίνακα στον οποίο ο χρήστης θέλει να βρει το νούμερο



18.

Στον κώδικα έχουμε προσθέσει μια συνάρτηση με όνομα interpolationSearch η οποία είναι ίδια με την binarySearch με μερικές αλλαγές. Αναλυτικότερα, αντί ο

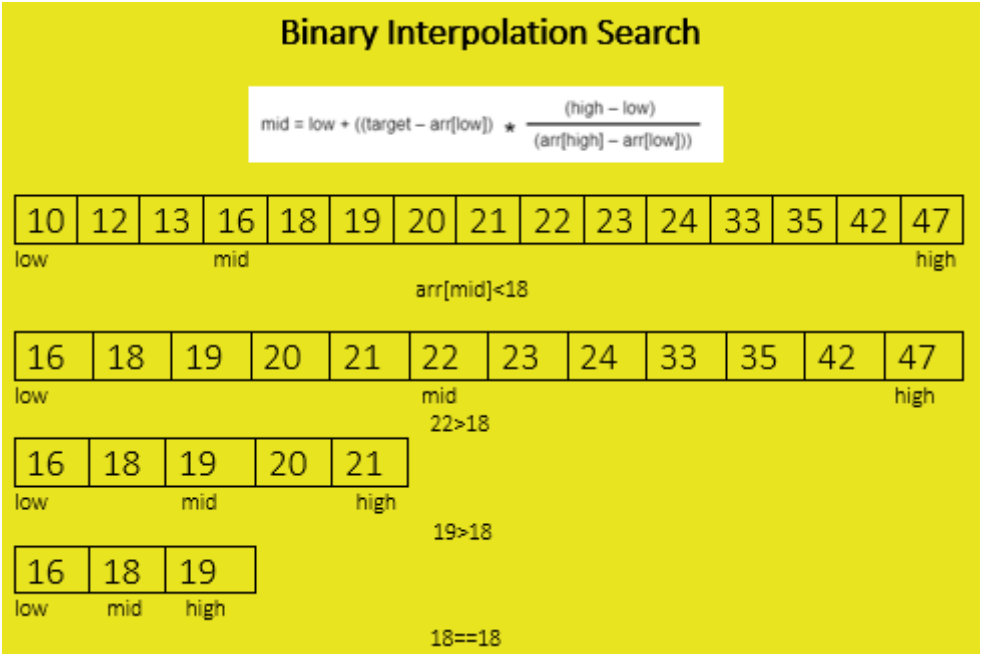
```
int mid = (low + high) / 2;
int pos = low + ((high - low) * (compareDates(searchDate, entries[low].Date))) /
        (compareDates(entries[high].Date, entries[low].Date));
```

πίνακας να διαιρείται στα 2, και να παίρνουμε το νούμερο στην μέση, ορίζουμε το pos(position) με τον συγκεκριμένο τύπο.

Για την υλοποίηση και των δύο αλγορίθμων απαιτείται η ταξινόμηση του πίνακα. Αν ο πίνακας δεν είναι ταξινομημένος ,κάθε φορά που θα συγκρίνεται η τιμή mid ή pos με την ημερομηνία που δήλωσε ο χρήστης, δεν θα μπορούμε να αποκλείσουμε την δεξιά ή την αριστερή μεριά του πίνακα καθώς μια πιθανή τιμή θα μπορεί να βρίσκεται εκεί. Επίσης, ο binary search έχει μέση απόδοση O(logn) ενώ ο Interpolation Search, O(log(logn)) όπου n είναι το μέγεθος του πίνακα. Αυτό συμβαίνει γιατί η αναζήτηση με διαδοχική διαίρεση, αν ο πίνακας είναι ομοιόμορφα κατανεμημένος, έχει προοπτική να εκτελεστεί πιο γρήγορα από την δυαδική αναζήτηση. Ωστόσο, σε περίπτωση που η κατανομή δεν είναι ομοιόμορφη (στην πλειοψηφία των περιπτώσεων), ο πιο γρήγορος αλγόριθμος είναι της δυαδικής αναζήτησης.

4) Binary Interpolation Search

Το Binary Interpolation Search, για την εκτέλεσή του απαιτεί ταξινομημένο πίνακα. Η εφαρμογή του είναι παρόμοια με την interpolation search, δηλαδή χρησιμοποιεί παρεμβολή (τον ίδιο τύπο με την αναζήτηση που αναφέραμε πιο πάνω) για να βρεί μια τιμή με το όνομα position (στον κώδικά μας έχει όνομα pos). Ύστερα συγκρίνουμε το αποτέλεσμα με το date που θέλουμε, και άμα είναι ίσα βρήκαμε την τιμή, αν είναι μεγαλύτερο απορρίπτουμε την δεξιά μεριά του πίνακα και αντίστοιχα, αν είναι μικρότερο απορρίπτουμε την αριστερή. Αμέσως μετά έχουμε έναν καινούριο πίνακα, οπότε σε αυτόν εφαρμόζουμε binary search. Δηλαδή διαιρούμε τον πίνακα στα 2, και συγκρίνουμε την τιμή που βρίσκεται στην μέση αυτού με την



τιμή της ημερομηνίας που είχε συμπληρώσει ο χρήστης. Με την ίδια μεθοδολογία με την interpolation search, απορρίπτουμε το αντίστοιχο κομμάτι του πίνακα, και επαναλαμβάνουμε την binary search.

🔍Εστιάζοντας στον **κώδικά** μας, για την εκτέλεση του Binary Interpolation Search, αρχικά ξαναχρησιμοποιούμε την quicksort, την partition και την swap, αμέσως μετά ορίζουμε μια συνάρτηση με το όνομα binaryInterpolationSearch. Στην οποία έχουμε σαν ορίσματα τα low(την χαμηλότερη τιμή του πίνακα), high(την μεγαλύτερη τιμή του πίνακα), searchDate(η ημερομηνία που είχε συμπληρώσει ο χρήστης), count(ένας χώρος αποθήκευσης, με χρήση counter). Μετά ορίζουμε το foundIndex = -1 το ποιο θα αλλάξει τιμή με την εύρεση της τιμής που ψάχνουμε και το *count = 0. Μετά χρειάζεται το while statement για να εξασφαλίσουμε ότι το low <= high και ότι το searchDate είναι μεγαλύτερο ή ίσο με το low και μικρότερο ή ίσο με το high . Ύστερα βρίσκουμε το pos, η τιμή που θα συγκριθεί με το searchDate.Μετά

```
int binaryInterpolationSearch(DataEntry *entries, int low, int high, const char *searchDate, int *count) {
    int foundIndex = -1;
    *count = 0;
    while (low <= high && compareDates(searchDate, entries[low].Date) >= 0 &&
           compareDates(searchDate, entries[high].Date) <= 0) {

        int pos = low + ((high - low) * (compareDates(searchDate, entries[low].Date))) /
                                        (compareDates(entries[high].Date, entries[low].Date));
        int comparison = compareDates(entries[pos].Date, searchDate);
        if (comparison == 0) {
            foundIndex = pos;
            low = pos + 1;
            (*count)++;
        }
        if (comparison < 0) {
            low = pos + 1;
        }
        else {
            high = pos - 1;
        }}
    return foundIndex;
}
```

εφαρμόζουμε μια σύγκριση του searchDate με το pos ορίζοντας το comparison το οποίο σύμφωνα με το αποτέλεσμα θα πάρει την τιμή -1,0 ή 1. Αν είναι 0, foundIndex = pos καθώς βρέθηκε η τιμή, αν είναι -1, απορρίπτουμε την αριστερή μεριά, οπότε το καινούριο low = pos + 1, και αντίστοιχα ,αν είναι 1, high = pos – 1. Στο τέλος επιστρέφει το foundIndex, που είναι είτε ο δείκτης, αν αυτός έχει βρεθεί είτε -1 στην αντίθετη περίπτωση.

Επιπροσθέτως *, για την βελτίωση του χρόνου χειρότερης περίπτωσης συμβουλευτήκαμε το βιβλίο των ‘’ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ’’ του Αθανάσιου Κ.Τσακαλίδη, από το οποίο πήραμε την ιδέα να τροποποιήσουμε κατάλληλα τον κώδικα ώστε στο while loop αντί να αυξάνεται το (*count)++ γραμμικά, θα ορίσουμε ένα I leap το οποίο θα αυξάνεται εκθετικά i = i * 2.*

Αναλυτικότερα, ορίσαμε μια νέα συνάρτηση με το όνομα ModifiedBIS, στην οποία οι αλλαγές που κάναμε σε σύγκριση με την binaryInterpolationSearch είναι οι εξής:

*ορίσαμε το ‘int i = 1;’ Και το αρχικοποιήσαμε ίσο με 1.

*αμέσως μετά τα ορίσματα του while loop ορίζουμε το pos ίσο με low + i, ώστε να υπολογίσουμε την επόμενη θέση.

*Μετά εξασφαλίζουμε ότι η ημερομηνία που ψάχνουμε βρίσκεται μέσα στο εύρος του πίνακα.

*If statement και αν η τιμή του pos είναι μικρότερο ή ίσο με το date που ψάχνουμε, ορίζουμε καινούριο low = pos και i = i * 2, αλλιώς high = pos – 1, και μετά από αυτή την αλλαγή, κάνουμε έξοδο από το λούπ αφού η υπολογισμένη ποσότητα είναι μεγαλύτερη.

```
    int pos = low + i;
    if (pos > high) {
        pos = high;}
    if (compareDates(entries[pos].Date, searchDate) <= 0) {
        low = pos;
        i = i * 2;
    } else {
        high = pos - 1;
        break; }}
while (low <= high) {
    int mid = (low + high) / 2;
    int comparison = compareDates(entries[mid].Date, searchDate);
    if (comparison == 0) {
        foundIndex = mid;
        (*count)++;
        break;
    } else if (comparison < 0) {
        low = mid + 1;
    } else {
        high = mid - 1; }}
```

Έπειτα η συνάρτηση λειτουργεί σαν την binary search, και επιστρέφει το foundIndex, δηλαδή τον δείκτη.

Ο χρόνος χειρότερης περίπτωσης του Binary Interpolation Search είναι O(logn) και συμβαίνει όταν ο πίνακας που θέλουμε να ψάξουμε δεν είναι ομοιόμορφα κατανεμημένος, δηλαδή οι διακυμάνσεις μεταξύ των τιμών είναι πολύ μεγάλες. Του ModifiedBIS είναι O(logn) και συμβαίνει όταν η τιμή που ψάχνουμε είναι είτε στο τέλος του πίνακα είτε στο μέσο του διαστήματος αναζήτησης κάθε επανάληψης είτε δεν υπάρχει. Και στους 2 χρόνους, το n είναι το πλήθος των εγγραφών στον πίνακα.

Part 2

Για την δημιουργία του κώδικα στο part 2 διατηρήσαμε κάποια κομμάτια ίδια από τον κώδικα του προηγούμενου μέρους. Πιο συγκεκριμένα, το header file με το όνομα dataentry.h, και όλα αυτά που συμπεριλαμβάνονται μέσα. Επιπλέον μερικά σημεία της main σαν λογική είναι ίδια και θα αναφερθούμε περιληπτικά σε αυτά. Αρχικά βάζουμε το πρόγραμμα να αναγνωρίζει το φάκελο με τα δεδομένα μας «effects.csv» και ορίζει το headers με σκοπό να μην διαβάζονται και οι τίτλοι του αρχείου. Ύστερα με την χρήση του while loop διαβάζει ο κώδικας τα στοιχεία με 2 τρόπους καθώς όπως και στην αρχή, έτσι και τώρα, δεν μπορούν να διαβαστούν κάποια σύμβολα που περιέχονται στο αρχείο. Επιπλέον, έχουμε δημιουργήσει ένα πρότυπο για το μενού με την χρήση choices και cases, για να είναι όσο φιλικό γίνεται προς τον χρήστη. Τέλος, σε οποιοδήποτε σημείο που το πρόγραμμα αποτύχει να εκτελέσει την δουλειά που θέλουμε να κάνει ή ο χρήστης επιλέξει μία επιλογή για την οποία δεν έχουμε διαμορφώσει κώδικα, το πρόγραμμα θα εκτυπώνει ένα αντίστοιχο μήνυμα.

```
typedef struct DataEntry {
    char Direction[20];
    int Year;
    char Date[11];
    char Weekday[20];
    char Country[20];
    char Commodity[20];
    char Transport_mode[20];
    char Measure[3];
    int Value;
    unsigned int Cumulative;
} DataEntry;
```

```
FILE* file = fopen("effects.csv", "r");
if (file == NULL) {
    printf("Error opening effects.csv\n");
    return 1;
}
```

```
char headers[256];
if (fgets(headers, sizeof(headers), file) == NULL) {
    printf("Error reading headers\n");
    fclose(file);
    return 1;}
```

```
printf("Invalid choice. Please try again.\n");
```

<u>Καθώς το πρόγραμμα είναι ολοκληρωμένο, δηλαδή έχουμε δημιουργήσει τον κώδικα που εκτελεί όλο το part 2 και έχουμε ενσωματώσει τα (Α),(Β) και (Γ), θα αναλυθεί ο κώδικας από πάνω προς τα κάτω ανά συναρτήσεις, με την main και με συμπληρώσεις, και όχι ανά κομμάτια, όπως είχε γίνει στο part 1.</u>

Ο κώδικάς μας κάνει include τις <stdio.h>, <stdlib.h> και <string.h> καθώς και τον header file μας "dataentry.h" ώστε να είναι δυνατόν να χρησιμοποιηθούν παρακάτω δηλώσεις χρήσιμες για την έξοδο- είσοδο δεδομένων, δυναμική δέσμευση-απελευθέρωση μνήμης, για την επεξεργασία

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dataentry.h"

#define MAX_SIZE 11 // Megeθos tou hash pinaka
```

αλφαριθμητικών και πάρα πολλά άλλα.

Επίσης από την αρχή έχουμε δεσμεύσει ένα μέγεθος για τον πίνακα hash που θα αναφερθεί πολύ αναλυτικά αργότερα.

Ορίζουμε στον κώδικάς μας έναν τύπο δομής με όνομα **BSTNode** , με σκοπό να δημιουργήσουμε έναν κόμβο σε ένα δυαδικό δέντρο αναζήτησης (με το αντίστοιχο όνομα). Μέσα η δομή έχει 3 μέλη, το record(ένα μέλος τύπου DataEntry), το left (δείκτης σε άλλο κόμβο, που εστιάζει στο αριστερό υπόδεντρο του κόμβου) και το right(δείκτης σε άλλο κόμβο, που εστιάζει στο δεξιό υπόδεντρο του κόμβου). Αυτό το struct έχει την κατάλληλη δομή ώστε να μπορέσει να αναπαραστήσει ένα BST, το record είναι ένας κόμβος με τα δεδομένα, και τα left και right οι υπόκομβοι.

```
typedef struct BSTNode {
    DataEntry record;
    struct BSTNode* left;
    struct BSTNode* right;
} BSTNode;
```

```
typedef struct HashNode {
    DataEntry record;
    struct HashNode* next;
} HashNode;
```

Ένας άλλος τύπος δομής που ορίζουμε είναι και ο **HashNode.** Έχει 2 μέλη, το record (μέλος του DataEntry, στον οποίο θα αποθηκεύονται τα δεδομένα στον κόμβο της κατακερματισμένης δομής), και το next(δείκτης σε άλλο κόμβο τύπου HashNode, που θα δείχνει στον επόμενο κόμβο στην ίδια κατακερματισμένη καταχώρηση). Κάθε κατακερματισμένη καταχώρηση περιέχει έναν ή περισσότερους κόμβους τύπου HashNode, και οι κόμβοι συνδέονται μέσω του πεδίου next, αυτό επιτρέπει την αποθήκευση πολλαπλών στοιχείων σε μια μόνο θέση του κατακερματισμένου πίνακα και την αντιμετώπιση συγκρούσεων (collisions) κατά την εισαγωγή δεδομένων.

Συναρτήσεις

Πιο κάτω δημιουργούμε τον πίνακα hash. Αρχικά δηλώνουμε έναν **πίνακα hashTable** τύπου HashNode* με μέγεθος MAX_SIZE(11). Ο πίνακας αυτός θα χρησιμοποιηθεί για την αποθήκευση κατακερματισμένων κόμβων τύπου HashNode. Μετά ορίζουμε την πρώτη μας συνάρτηση με το όνομα hashFuction, με παράμετρο το date και έναν αριθμό m (το μέγεθος του πίνακα). Ύστερα ορίζουμε την asciiSum, που την αρχικοποιούμε ίση με 0, και είναι χρήσιμη γιατί θα αποθηκεύουμε εκεί το άθροισμα των ASCII κωδίκων των χαρακτήρων του date. Παρακάτω, με την χρήση for loop, η μεταβλητή asciiSum αυξάνεται κατά την τιμή του ακέραιου αντίστοιχου του κάθε χαρακτήρα.Και τέλος, η συνάρτηση επιστρέφει το υπόλοιπο της διαίρεσης του asciiSum με το m. Αυτό εξασφαλίζει ότι ο κατακερματισμός θα είναι μια τιμή μεταξύ 0 και m-1, ώστε να είναι έγκυρος δείκτης για τον πίνακα hashTable.

```
HashNode* hashTable[MAX_SIZE];

int hashFunction(const char* date, int m) {
    int asciiSum = 0;
    for (int i = 0; i < strlen(date); i++) {
        asciiSum += (int)date[i];
    }
    return asciiSum % m;
}
```

Η συνάρτηση **insertRecord** χρησιμοποιείται για την εισαγωγή record στον πίνακα hash. Για την υλοποίηση αυτού, έχει ως παραμέτρους το date και τους ακέραιους value και m.Έπειτα θέλουμε να κάνουμε κατακερματισμό του date με την χρήση της συνάρτησης hashFunction(η πιο πάνω), και αποθήκευσή του στην μεταβλητή key. Κατόπιν δημιουργούμε καινούργιο κόμβο με το όνομα newNode με τη χρήση της malloc για την δέσμευση μνήμης. Αρχικοποιούμε τον νέο κόμβο με τα δεδομένα του date και του value. Το πεδίο next του νέου κόμβου ορίζεται ως NULL. Μετά με έναν γρήγορο έλεγχο για το αν η θέση του πίνακα hashTable[key] είναι κενή, ο νέος κόμβος γίνεται η πρώτη καταχώρηση στην θέση αυτή, αλλιώς, ο νέος κόμβος τοποθετείται στην αρχή της αλυσίδας, με το next να δείχνει τον προηγούμενο κόμβο.

```
void insertRecord(const char* date, int value, int m) {
    int key = hashFunction(date, m);
    HashNode* newNode = (HashNode*)malloc(sizeof(HashNode));
    strcpy(newNode->record.Date, date);
    newNode->record.Value = value;
    newNode->next = NULL;
    if (hashTable[key] == NULL) {
        hashTable[key] = newNode;
    } else {
        newNode->next = hashTable[key];
        hashTable[key] = newNode;
    }
}
```

Συναρτήσεις

Στον κώδικά μας προσθέτουμε άλλη μία συνάρτηση, την **searchRecordHash,** η οποία ψάχνει και εμφανίζει τιμές εγγραφών στον hashTable **με βάση το date** και το m. Ορίζουμε την μεταβλητή key που υπολογίζει τον κατακερματισμό του date με τη

```
int searchRecordHash(const char* date, int m) {
    int key = hashFunction(date, m);
    HashNode* currentNode = hashTable[key];
    int foundRecords = 0;
    while (currentNode != NULL) {
        if (strcmp(currentNode->record.Date, date) == 0) {
            printf("Value for Date %s: %d\n", date, currentNode->record.Value);
            foundRecords++; }
        currentNode = currentNode->next;}
    if (foundRecords == 0) {
        printf("No records found for Date %s\n", date);}
    return key;  }
```

χρήση της συνάρτησης hashFunction. Αμέσως μετά δημιουργούμε έναν δείκτη με το όνομα currentNode τύπου HashNode που αρχικά δείχνει στον πρώτο κόμβο της αλυσίδας στη θέση hashTable[key]. Στην επόμενη φάση αρχικοποιούμε την μεταβλητή foundRecords στο 0 (θα χρησιμοποιηθεί για να μετρήσει τον αριθμό των εγγραφών που βρέθηκαν).Ύστερα, το while loop εκτελείται μέχρι να φτάσει στο τέλος της αλυσίδας, που σημαίνει ότι έχουν εξεταστεί όλοι οι κόμβοι. Κατόπιν ,εντός του βρόγχου, ελέγχεται αν το date του τρέχοντος κόμβου currentNode είναι ίδιο με το date που έχει δοθεί. Αν είναι ίδια, τότε εμφανίζεται η τιμή Value του κόμβου με τη χρήση της printf. Ταυτόχρονα, αυξάνεται η μεταβλητή foundRecords κατά 1 για να καταγράψει τον αριθμό. Αν ο μετρητής foundRecords παραμείνει μηδενικός, τότε δεν βρέθηκαν εγγραφές για τη συγκεκριμένη ημερομηνία, και εκτυπώνεται το αντίστοιχο μήνυμα. Τέλος, η συνάρτηση επιστρέφει τον κατακερματισμό (key) ως δείκτη στον πίνακα κατακερματισμού για τις ευρεθείσες εγγραφές.

Στην επόμενη φάση, ορίζουμε μια βοηθητική συνάρτηση με το όνομα **insertBST**, που χρησιμοποιείται για την εισαγωγή μιας νέας εγγραφής στο BST. Δέχεται 3 ορίσματα, το root(δείκτης στην ρίζα του BST και ενώ στην αρχή είναι ίσος με NULL,

```
BSTNode* insertBST(BSTNode* root, DataEntry entry, int compareByDate) {
    if (root == NULL) {
        BSTNode* newNode = (BSTNode*)malloc(sizeof(BSTNode));
        newNode->record = entry;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;}
    if (compareByDate) {
        if (strcmp(entry.Date, root->record.Date) < 0) {
            root->left = insertBST(root->left, entry, compareByDate);
        } else {root->right = insertBST(root->right, entry, compareByDate);}
     } else {
        if (entry.Value < root->record.Value) {
            root->left = insertBST(root->left, entry, compareByDate);
        } else {root->right = insertBST(root->right, entry, compareByDate);}}
    return root;}
```

ύστερα θα δείχνει στην υπάρχουσα ρίζα), το entry (η εγγραφή που θέλουμε να εισάγουμε) και το compareByDate(ακέραιος που υποδηλώνει αν η σύγκριση των εγγραφών γίνεται με βάση την ημερομηνία (1) ή την τιμή (0)). Στην συνέχεια, η συνάρτηση insertBST υλοποιεί την αναδρομική εισαγωγή μιας νέας εγγραφής στο BST με τη σωστή τοποθέτηση στο δέντρο και αν η ρίζα (root) είναι NULL, τότε δημιουργείται ένας νέος κόμβος (newNode) και ανατίθεται σε αυτόν η νέα εγγραφή (entry),(ο νέος κόμβος αρχικά δεν έχει παιδιά και επιστρέφεται ως η νέα ρίζα του BST) ενώ αν η ρίζα δεν είναι NULL, τότε γίνεται σύγκριση της νέας εγγραφής (entry) με την εγγραφή που βρίσκεται στη ρίζα του BST. Αν compareByDate είναι 1, γίνεται σύγκριση βάσει της ημερομηνίας, διαφορετικά γίνεται σύγκριση βάσει της τιμής. Αν η νέα εγγραφή είναι μικρότερη, τότε καλείται αναδρομικά η insertBST για την εισαγωγή της εγγραφής στο αριστερό υποδέντρο Αν η νέα εγγραφή είναι μεγαλύτερη, τότε καλείται αναδρομικά η insertBST για την εισαγωγή της εγγραφής στο δεξί υποδέντρο. Τέλος, η συνάρτηση επιστρέφει τη ρίζα του BST μετά την εισαγωγή της νέας εγγραφής.

Στην συνέχεια, ορίζουμε την συνάρτηση **searchRecordBST**, που χρησιμοποιείται για να αναζητήσει μια εγγραφή σε ένα δυαδικό δέντρο αναζήτησης (Binary Search Tree - BST), **βάσει της ημερομηνίας**. Η συνάρτηση αυτή δέχεται 4 ορίσματα, το root (δείκτης στη ρίζα του BST), το date (η ημερομηνία που αναζητούμε) και το compareByDate (ακέραιος που υποδηλώνει αν η σύγκριση των εγγραφών γίνεται με

```
void searchRecordBST(BSTNode* root, const char* date, int compareByDate) {
    if (root == NULL) {
        printf("Record not found for Date %s\n", date);
        return;}
    int compare;
    int found = 0;
```

βάση την ημερομηνία (1) ή την τιμή (0)).Ύστερα χρησιμοποιούμε το while loop, για να αναζητηθεί η εγγραφή με την επιθυμητή ημερομηνία στο BST. Κατά τη διάρκεια της αναζήτησης, η σύγκριση της εγγραφής με την επιθυμητή ημερομηνία γίνεται με βάση τον ακέραιο compareByDate. Εάν η ρίζα (root) είναι NULL, τότε δεν βρέθηκε εγγραφή με τη συγκεκριμένη ημερομηνία και εκτυπώνεται το ανάλογο μήνυμα. Σε κάθε επανάληψη, γίνεται σύγκριση της επιθυμητής ημερομηνίας (date) με την ημερομηνία της τρέχουσας εγγραφής στον κόμβο root. Αν οι ημερομηνίες είναι ίσες, τότε βρέθηκε η εγγραφή και εκτυπώνεται η τιμή που συνοδεύει την ημερομηνία. Επιπλέον, ο δείκτης root μετακινείται στο δεξιό υποδέντρο, καθώς αναζητούμε τυχόν επιπλέον εγγραφές με την ίδια ημερομηνία. Αν οι ημερομηνίες διαφέρουν, τότε ανάλογα με το αποτέλεσμα της σύγκρισης, ο δείκτης root μετακινείται αριστερά (αν η επιθυμητή ημερομηνία είναι μικρότερη) ή δεξιά (αν η επιθυμητή ημερομηνία είναι μεγαλύτερη) στο BST. Αν η επανάληψη ολοκληρωθεί χωρίς να βρεθεί η εγγραφή με την επιθυμητή ημερομηνία, τότε εκτυπώνεται το μήνυμα αντίστοιχο μήνυμα.

Συναρτήσεις

Ορίζουμε άλλη μία συνάρτηση, με το όνομα **inorderTraversalToFile**, η οποία πραγματοποιεί μια διάσχιση inorder σε ένα δυαδικό δέντρο αναζήτησης (Binary Search Tree - BST) και εγγράφει τα δεδομένα σε ένα αρχείο.Η συνάρτηση μας δέχεται δύο ορίσματα, το root (δείκτης στη ρίζα του BST, από όπου ξεκινά η διάσχιση) και το outputFile (δείκτης σε ανοιχτό αρχείο, όπου θα εγγραφούν τα

```
void inorderTraversalToFile(BSTNode* root, FILE* outputFile) {
    if (root == NULL) {
        return;}
    inorderTraversalToFile(root->left, outputFile);
    fprintf(outputFile, "Direction: %s, Year: %d, Date: %s, Weekday: %s, Country: %s, Commodity: %
            root->record.Direction, root->record.Year, root->record.Date, root->record.Weekday,
            root->record.Country, root->record.Commodity, root->record.Transport_mode,
            root->record.Measure, root->record.Value, root->record.Cumulative);
    inorderTraversalToFile(root->right, outputFile);
    }
```

δεδομένα) .Η διάσχιση inorder γίνεται αναδρομικά. Σε κάθε βήμα, η συνάρτηση καλεί τον εαυτό της για το αριστερό υποδέντρο, στη συνέχεια εγγράφει τα δεδομένα του τρέχοντος κόμβου στο αρχείο και, τέλος, καλεί τον εαυτό της για το δεξί υποδέντρο. Η εγγραφή στο αρχείο γίνεται με τη χρήση της συνάρτησης fprintf. Τα δεδομένα που εγγράφονται περιλαμβάνουν όλα τα πεδία της δομής DataEntry, χρησιμοποιώντας τα %s, %d και %u για τους διάφορους τύπους δεδομένων.Πιο αναλυτικά, η διάσχιση inorder εξασφαλίζει ότι τα δεδομένα θα εγγραφούν στο αρχείο με αύξουσα σειρά βάσης της τιμής τους. Αυτό συμβαίνει επειδή πρώτα διασχίζονται τα αριστερά υποδέντρα, έπειτα εγγράφονται τα δεδομένα των κόμβων και τέλος διασχίζονται τα δεξιά υποδέντρα. Έτσι, τα δεδομένα εμφανίζονται στο αρχείο με αύξουσα σειρά βάση της τιμής τους, όπως ορίζεται από το BST.

Για να είναι δυνατόν να απεικονίσουμε το BST με τα δεδομένα του και να τα εγγράψουμε σε ένα αρχείο, δημιουργήσαμε την συνάρτηση **displayBST.** Η ίδια κάνει έναν ελέγχο για το αν το δυαδικό δέντρο είναι κενό με την χρήση του if statement, και να είναι εμφανίζει το αντίστοιχο μήνυμα, και τελειώνει η συνάρτηση. Αν δεν

```
void displayBST(BSTNode* root) {
    if (root == NULL) {
        printf("BST is empty.\n");
        return;}
    FILE* outputFile = fopen("bst_display.txt", "w");
    if (outputFile == NULL) {
        printf("Error opening file for writing.\n");
        return;}
    inorderTraversalToFile(root, outputFile);
    fclose(outputFile);
    printf("BST displayed successfully and written to bst_display.txt.\n");
}
```

είναι, η συνάρτηση συνεχίζει ανοίγοντας το αρχείο εξόδου με το όνομα "bst_display.txt" για να κάνει την εγγραφή. Έπειτα, εκτελεί την inorderTraversalToFile

και εγγράφει τα δεδομένα κάθε κόμβου στο αρχείο εξόδου, εξασφαλίζοντας ότι τα δεδομένα θα εμφανίζονται σε αύξουσα σειρά βάσης της τιμής της ημερομηνίας. Κια τέλος κλείνει ο αρχείο, εκτυπώνοντας στην οθόνη ότι ήταν δυνατή η εκτέλεση αυτής της συνάρτησης.

```
if (!found) {
    printf("Record not found for Date %s\n", date);
}
```

Ο χρήστης μπορεί να θελήσει να αλλάξει το value σε μια συγκεκριμένη ημερομηνία στο BST, για αυτόν τον λόγο ορίσαμε την **modifyValue**. Η συνάρτηση παίρνει σαν ορίσματα το root (δείκτης στη ρίζα του BST), το date (η ημερομηνία που αναζητούμε) και το newValue(η νέα τιμή) και αρχικά ελέγχει αν το δυαδικό δέντρο

```
void modifyValue(BSTNode* root, const char* date, int newValue) {
    if (root == NULL) {
        printf("Record not found for Date %s\n", date);
        return;}
    int compare = strcmp(date, root->record.Date);
    if (compare == 0) {
        root->record.Value = newValue;
        printf("Value for Date %s modified to %d\n", date, newValue);
    } else if (compare < 0) {
        modifyValue(root->left, date, newValue);
    } else {
        modifyValue(root->right, date, newValue);}}
```

είναι κενό, άρα δεν μπορεί να τροποποιηθεί η ημερομηνία, άρα εμφανίζεται αντίστοιχο μήνυμα. Στην αντίθετη περίπτωση, συγκρίνεται η ημερομηνία date με του κόμβου που είμαστε, Αν είναι ίσες τότε εντοπίζει τον σωστό κόμβο και αλλάζει το value του κόμβου σε newValue και εκτυπώνει ένα μήνυμα επιβεβαίωσης ότι η τιμή έχει τροποποιηθεί με επιτυχία. Αν οι ημερομηνίες δεν είναι ίσες, τότε συνεχίζει την αναζήτηση ανάλογα με τη σειρά ταξινόμησης του BST. Αν το date είναι μικρότερο από την ημερομηνία του τρέχοντος κόμβου, τότε συνεχίζεται η αναζήτηση στο αριστερό υποδέντρο. Διαφορετικά, συνεχίζεται η αναζήτηση στο δεξί υποδέντρο. Αυτή η διαδικασία επαναλαμβάνεται μέχρι να βρεθεί ο κατάλληλος κόμβος ή να φτάσει σε ένα NULL υποδέντρο.

Αντίστοιχη συνάρτηση χρειαζόμαστε για την αλλαγή τιμών μιας συγκεκριμένης ημερομηνίας αλλά στον πίνακα hash, και την ονομάσαμε **modifyRecordValue** που

```
void modifyRecordValue(const char* date, int size) {
    int index = searchRecordHash(date, size);
    if (index != -1) {
        printf("Enter the new value: ");
        int newValue;
        scanf("%d", &newValue);
        hashTable[index]->record.Value = newValue;
        printf("Record value modified successfully.\n");
    } else {
        printf("No record found for the given date.\n");}}
```

παίρνει σαν ορίσματα το date και το newValue. Αρχικά καλεί τη συνάρτηση searchRecordHash για να αναζητήσει την ημερομηνία στον πίνακα hash και επιστρέφει τον δείκτη (index) του κόμβου που αντιστοιχεί στην ημερομηνία. Αν ο δείκτης είναι ίσος με το -1, υποδηλώνεται ότι η ημερομηνία δε βρέθηκε, αν είναι διάφορος από το -1, τότε ζητά από τον χρήστη να εισάγει τη νέα τιμή (newValue) για τον κόμβο.Στη συνέχεια, το value του κόμβου στον πίνακα hash τροποποιείται και αναθέτεται νέα τιμή στο πεδίο hashTable[index]->record.Value. Αυτό σημαίνει ότι η τιμή του κόμβου στον πίνακα hash αλλάζει με τη νέα τιμή που συμπλήρωσε ο χρήστης. Και τέλος, εκτυπώνεται ένα μήνυμα επιβεβαίωσης ότι η τιμή του κόμβου έχει τροποποιηθεί με επιτυχία. Αν η ημερομηνία δε βρέθηκε στον πίνακα hash, τότε εκτυπώνεται αντίστοιχο μήνυμα.

Η παρακάτω συνάρτηση με το όνομα **deleteNode** θα δίνει την δυνατότητα στον χρήστη να διαγράφει εντελώς κόμβους με dates από το BST. Στην αρχή γίνεται ένας έλεγχος για τον αν ο root είναι κενός ή όχι, αν είναι, εμφανίζεται αντίστοιχο μήνυμα, αλλιώς συνεχίζεται η συνάρτηση. Χρησιμοποιούμε το compare για την σύγκριση της date με την ημερομηνία του κόμβου.

```
int compare = strcmp(date, root->record.Date);
if (compare < 0) {
    root->left = deleteNode(root->left, date);
} else if (compare > 0) {
    root->right = deleteNode(root->right, date);
} else {=
    if (root->left == NULL) {
        BSTNode* temp = root->right;
        free(root);
        return temp;
    } else if (root->right == NULL) {
        BSTNode* temp = root->left;
        free(root);
        return temp;}
```

Αν η strcmp είναι αρνητική, τότε η διαγραφή θα γίνει στο αριστερό υποδέντρο, και καλείται η deleteNode αναδρομικά για το αριστερό υποδέντρο. Αν είναι θετική, θα γίνει η αντίστοιχη διαδικασία στο δεξί υποδέντρο. Αν είναι ίση με 0, τότε βρέθηκε η ημερομηνία και ο συγκεκριμένος κόμβος πρέπει να διαγραφεί, και από εκεί παίρνουμε 2 περιπτώσεις:

- ➔ Αν ο κόμβος που πρέπει να διαγραφεί δεν έχει παιδιά ή έχει μόνο ένα παιδί, τότε απλά αντικαθίσταται από το παιδί του ή το παιδί που έχει, ενώ ο αρχικός κόμβος διαγράφεται και μετά, επιστρέφεται το παιδί που αντικατέστησε τον κόμβο που διαγράφηκε.
- ➔ Αν ο κόμβος που πρέπει να διαγραφεί έχει δύο παιδιά, τότε εντοπίζεται ο κόμβος temp που αποτελεί τον επόμενο κόμβο στη σειρά (βάσει της ταξινόμησης) και αντιγράφεται η εγγραφή του στον κόμβο root. Στη συνέχεια, η

```
    BSTNode* temp = root->right;
    while (temp->left != NULL)
        temp = temp->left;
    root->record = temp->record;
    root->right = deleteNode(root->right, temp->record.Date);}
return root;}
```

deleteNode καλείται για να διαγράψει τον κόμβο temp από το δεξί υποδέντρο του root.

Στο τέλος επιστρέφεται η τιμή root του BST.

Κατόπιν δημιουργούμε τις συναρτήσεις **findMinValue** και **findMaxValue** για να είναι εφικτό να βρεθεί η ελάχιστη και η μέγιστη τιμή στα BST. Αναλυτικότερα:

η πρώτη δέχεται έναν κόμβο του BST ως παράμετρο και ελέγχει αν είναι κενός ή αν έχει αριστερό παιδί, Αν ισχύει, τότε αυτός ο κόμβος έχει την ελάχιστη τιμή, άρα επιστρέφεται στο τέλος. Αντίθετα, αν ο node έχει αριστερό παιδί , με την χρήση του if statement καλείται ξανά η συνάρτηση. Οπότε τελικά επιστρέφεται η ελάχιστη τιμή.

```
BSTNode* findMinValue(BSTNode* node) {
    if (node == NULL || node->left == NULL)
        return node;
    return findMinValue(node->left);}
BSTNode* findMaxValue(BSTNode* node) {
    if (node == NULL || node->right == NULL)
        return node;
    return findMaxValue(node->right);}
```

Και η δεύτερη λειτουργεί με αντίστοιχο τρόπο με την findMinValue, ωστόσο αντί να εστιάζει στο αριστερό παιδί, εκτελεί τις ίδιες λειτουργίες για το δεξί.

Για την διαγραφή ενός συγκεκριμένου αρχείου από τον πίνακα Hash με βάση την ημερομηνία (που δίνεται και ως όρισμα key), θα χρησιμοποιηθεί η συνάρτηση **deleteRecordHash**. Αρχικά υπολογίζει το hash του key χρησιμοποιώντας τη συνάρτηση hashFunction. Το hash αυτό αντιστοιχεί στον δείκτη του πίνακα κατακερματισμού (hash table) όπου μπορεί να βρίσκεται το αρχείο που θέλουμε να διαγράψουμε.Ξεκινάει από τον κόμβο που βρίσκεται στην αρχή της συνδεδεμένης λίστας στο hash table .Αμέσως μετά πραγματοποιεί μια αναζήτηση στη συνδεδεμένη λίστα για να βρει το αρχείο με το key που δόθηκε. Αυτό γίνεται συγκρίνοντας τις ημερομηνίες των αρχείων με τη ημερομηνία που έχει δοθεί.Αν βρεθεί το αρχείο, πραγματοποιείται η διαγραφή του από τη συνδεδεμένη λίστα.Αν το αρχείο είναι το πρώτο στοιχείο της λίστας (δηλαδή prevNode είναι NULL), τότε ο δείκτης του πίνακα hash για το συγκεκριμένο hash αλλάζει και δείχνει στο επόμενο αρχείο της λίστας (αν υπάρχει).

```
void deleteRecordHash(const char* key) {
    int index = hashFunction(key, MAX_SIZE);
    HashNode* currentNode = hashTable[index];
    HashNode* prevNode = NULL;
```

```
    while (currentNode != NULL) {
        if (strcmp(currentNode->record.Date, key) == 0) {
            if (prevNode == NULL) {
                hashTable[index] = currentNode->next;
            } else {
                prevNode->next = currentNode->next;}
            free(currentNode);
            printf("Record with date '%s' deleted successfully.\n", key);
            return;}
```

Αν το αρχείο δεν είναι το πρώτο στοιχείο, τότε ο δείκτης prevNode αλλάζει και δείχνει στο επόμενο αρχείο μετά το αρχείο που θέλουμε να διαγράψουμε και αποδεσμεύεται η μνήμη που είχε δεσμευθεί για το αρχείο που διαγράφηκε. Αν το αρχείο βρεθεί και διαγραφεί, εκτυπώνεται ένα μήνυμα για να επιβεβαιωθεί η επιτυχής διαγραφή του. Αν το αρχείο δε βρεθεί, εκτυπώνεται στην οθόνη το αντίστοιχο μήνυμα για να ενημερωθεί ο χρήστης ότι το αρχείο δεν βρέθηκε στον πίνακα hash.

```
        prevNode = currentNode;
        currentNode = currentNode->next;}
    printf("Record with date '%s' not found.\n", key);}
```

Μετά από αυτή την προετοιμασία με τις 13 συναρτήσεις, είμαστε στην θέση να μπορέσουμε να αναλύσουμε την main συνάρτησή μας.

Συνάρτηση **Main**

Αν και ένα κομμάτι της main έχει αναφερθεί στην αρχή του παρτ 2, εδώ θα τα δούμε την συνέχεια αυτής της ανάλυσης.

```
char line[256];
BSTNode* bstRoot = NULL;
int choice, compareByDate = 0;
```

Δημιουργούμε τις μεταβλητές bstRoot τύπου BSTNode* που δείχνει στη ρίζα ενός δυαδικού δέντρου αναζήτησης (BST), και compareByDate τύπου int που έχει αρχική τιμή 0.

Μετά με τη χρήση ενός while loop δημιουργείται ένα αντικείμενο entry τύπου DataEntry, στο οποίο αποθηκεύονται τα δεδομένα από κάθε γραμμή του αρχείου. Παράλληλα, γίνεται έλεγχος για τη σωστή ανάγνωση των δεδομένων. Αν το αποτέλεσμα της συνάρτησης sscanf δεν είναι 10, τότε εκτυπώνεται στην οθόνη το αντίστοιχο μήνυμα και το πρόγραμμα τερματίζεται με επιστροφή της τιμής 1. ‘Επειτα πραγματοποιείται η εισαγωγή του entry στο BST και στο hash table μέσω των συναρτήσεων insertBST και insertRecord (περισσότερες λεπτομέρειες στην αρχή του παρτ 2). Και στο τέλος, αφού ολοκληρωθεί το λουπ κλείνει το αρχείο με την fclose.

Στην επόμενη φάση εμφανίζεται ένα μενού στον χρήστη που τον ρωτάει με ποιόν τρόπο θέλει να φορτωθούν τα δεδομένα. Σε μορφή Binary Search Tree ή σε Hash Table.

*Αν επιλέξει την πρώτη μορφή, αμέσως μετά εμφανίζεται ένα δεύτερο μενού που τον ρωτάει αν θέλει η ταξινόμηση να γίνει με βάση το date ή το value.

```
case 1:
    displayBST(bstRoot);
    // Open the file automatically
    system("start notepad bst_display.txt");
    break;
case 2: {
    char date[11];
    printf("Enter the Date: ");
    scanf("%s", date);
    searchRecordBST(bstRoot, date, compareByDate);
    break;
}
case 3: {
    char date[11];
    int newValue;
    printf("Enter the Date: ");
    scanf("%s", date);
    printf("Enter the new Value: ");
    scanf("%d", &newValue);
    modifyValue(bstRoot, date, newValue);
    break;
}
case 4: {
    char date[11];
    printf("Enter the Date: ");
    scanf("%s", date);
    bstRoot = deleteNode(bstRoot, date);
    break;
}
default:
    printf("Invalid choice. Please try again.\n");
```

**Αν επιλεγεί η πρώτη επιλογή θα εμφανιστεί άλλο ένα μενού (3ο σε αριθμό) που τον ρωτάει τι κινήσεις θέλει να εκτελέσει πάνω στον BST: να απεικονιστεί το δέντρο με τις επικεφαλίδες του με την συνάρτηση displayBST, να αναζητηθεί από τον χρήστη μια τιμή value από ένα συγκεκριμένο date με την συνάρτηση searchRecordBST ,να τροποποιηθεί (modifyValue(bstRoot, date, newValue)ή να διαγραφεί (deleteNode) το περιεχόμενο που αντιστοιχεί σε ένα συγκεκριμένο date και τέλος να γίνει έξοδος από την εφαρμογή.

**Αν επιλέξει την 2η επιλογή θα εμφανιστεί το 3ο μενού που ρωτάει τον χρήστη τι ενέργειες θέλει εκτελέσει: την

εύρεση των ημερών ή της μέρας με την ελάχιστη ή την μέγιστη τιμή value και τέλος να γίνει έξοδος από το πρόγραμμα.

```
    if (minNode != NULL) {
        printf("Minimum value in BST: %d\n", minNode->record.Value);
        printf("Date of the minimum value: %s\n", minNode->record.Date);
    } else {
        printf("The BST is empty.\n");}
else if (choice == 2) {
    BSTNode* maxNode = findMaxValue(bstRoot);
    if (maxNode != NULL) {
        printf("Maximum value in BST: %d\n", maxNode->record.Value);
        printf("Date of the maximum value: %s\n", maxNode->record.Date);
    } else {
        printf("The BST is empty.\n");}
    }else if (choice == 3)
        break;
```

*Αν επιλέξει την 2η μορφή, σε πίνακα hash, το πρόγραμμα εμφανίζει ένα 2ο μενού που ρωτάει τον χρήστη ποια κίνηση θέλει να εκτελέσει: να αναζητήσει ένα value με βάση το date που δίνεται από τον χρήστη (με την συνάρτηση searchRecordHash),να τροποποιήσει ένα value σε ένα date (με την συνάρτηση modifyRecordValue) ή να διαγράψει μια εγγραφή από τον πίνακα βάσει date που δίνεται από τον χρήστη (deleteRecordHash)ή να κάνει έξοδο από το πρόγραμμα

Αντιμετώπιση Προβλημάτων

Προφανώς και δεν γίνεται όλη αυτή η προσπάθεια να μην είχε και διάφορα σκαμπανεβάσματα και δυσκολίες κατά την διάρκεια της δημιουργίας όλων αυτών των προγραμμάτων. Φτάσαμε ,λοιπόν, στο σημείο που καλούμαστε να αναφερθούμε και στα προβλήματα που αντιμετωπίσαμε.

Αρχικά, στο part 1 θα σταθούμε στους «πονοκέφαλους» που εμφανίστηκαν ανά ερώτημα. Όσον αφορά την counting sort και την merge sort, έχουμε να κάνουμε με δύο απλούς σε υλοποίηση αλγορίθμους, επομένως είναι φυσικό να μην δυσκολευτήκαμε πάρα πολύ. Έχουμε ήδη αναφερθεί στο σφάλμα που εμφάνιζε ο κώδικας μας κατά το διάβασμα του csv file και συγκεκριμένα στην στήλη measure.

Επίσης, πριν καταλήξουμε στην τρέχουσα μορφή του κώδικα, δυσκολευόματσταν να τυπώσουμε όλα τα ταξινομημένα στοιχεία του αρχείου γιατί δεν χωρούσαν στο command prompt που εμφανιζόταν όταν τρέχαμε τον κώδικα μας, εξού και η απόφαση να τυπώνεται το αποτέλεσμα του sort σε txt file.

Ακόμη, είχαμε ορίσει ως int το Cumulative αλλά εκτύπωνε αρνητικούς αριθμούς ή αριθμούς που δεν αντιστοιχούσαν στο αρχείο, οπότε πήραμε την απόφαση να το ορίσουμε με unsigned int και λύθηκε το πρόβλημα.

Προχωράμε τώρα στο 2^{ο} ερώτημα που αφορά heap sort και quicksort. Έγιναν αρκετά λάθη κατά την υλοποίηση του κώδικα είναι φυσικό να μην θυμόμαστε λεπτομερώς, ωστόσο, κάποια κύρια προβλήματα που μας ταλαιπώρησαν είναι τα παρακάτω:

1) Δεν γινόταν σωστή εκτύπωση και ταξινόμηση των στοιχείων, οπότε κάναμε debugging και προσθέσαμε στον κώδικα μας εντολές οι οποίες εκτύπωναν που γινόταν το λάθος και πόσες φορές (κομμάτι του κώδικα το οποίο αφαιρέθηκε μετά την διόρθωση του σφάλματος που το προκαλούσε). Το λάθος μας ,λοιπόν, ήταν πως εκ παραδρομής στην σύγκριση των cumulative δίναμε DataEntry ενώ αυτή δεχόταν int
2) Ένα, επίσης, λάθος για το οποίο δεν βρέθηκε ποτέ λύση είναι η εκτύπωση της λέξης Tonnes στο measure, η οποία άλλοτε τυπώνεται σωστά και άλλη με διάφορα «κινέζικα» αντί για τα σωστά γράμματα π.χ. Tonn*&

Συνεχίζουμε τώρα στο 3^{ο} και στο 4^{ο} ερώτημα τα οποία θα αναλύσουμε μαζί καθώς δεν παρουσιάζουν πολλές διάφορες στα προβλήματα.

Θεωρούμε απαραίτητο σε αυτό το σημείο να επισημάνουμε πως κρίθηκε κατάλληλο με βάση την εκφώνηση να τυπώνονται όλες οι περιπτώσεις μιάς ημερομηνίας.

Αντιμετώπιση Προβλημάτων

Κύριο μέλημα μας ήταν να καταφέρουμε να κάνουμε το πρόγραμμα μας να τις εκτυπώνει όλες. Αυτό , όμως, έμελλε να είναι και το σημείο που μας δυσκόλεψε περισσότερο. Ήταν εύκολα αντιληπτό πως χρειαζόμασταν ένα loop ούτως ώστε το αντίστοιχο search να επαναλαμβάνεται μέχρι να εμφανιστούν όλα τα instances μιας ημερομηνίας με τις διαφορετικές τιμές.

Διορθώθηκε, εν τέλει, και αυτό το πρόβλημα καθώς βρέθηκε το σωστό for loop και μέσω debugging και με τη χρήση των λεγόμενων «δικλείδων ασφαλείας» εντοπίσαμε και τα σημεία και τις γραμμές του κώδικα που έφταιγαν.

Ή τουλάχιστον έτσι πιστεύαμε.

Ο λόγος που ταλαιπωρηθήκαμε τόσο ήταν πως είχαμε ξεχάσει ότι για να λειτουργήσει η αναζήτηση έπρεπε να ταξινομηθεί ο πίνακας μας. Αλλιώς, πως θα έβρισκαν οι αλγόριθμοι που κάνουν συγκρίσεις από το μικρότερο στο μεγαλύτερο, την ημερομηνία που ψάχναμε, όσες φορές αυτή εμφανιζόταν.

Αποφασίσαμε τότε να χρησιμοποιήσουμε τον quicksort ο οποίος φαινομενικά έμοιαζε ο κατάλληλος αλγόριθμος για την ταξινόμηση, αλλά η αλήθεια είναι πως όπως θα διαπιστώσετε και εσείς χρείαζεται αρκετό χρόνο για να ολοκληρωθεί, χωρίς όμως το αποτέλεσμα να μην είναι το επιθυμητό.

Δυστυχώς, δεν βρήκαμε κάποιον τρόπο για να βελτιώσουμε αυτό τον χρόνο, αλλά αναγνωρίζουμε τουλάχιστον ότι πετύχαμε τον αρχικό μας στόχο.

Φτάνουμε τώρα και στο 2^{ο} μέρος του Project. Αναμφισβήτητα, έχουμε να κάνουμε με ένα πολύ πιο δύσκολο πρόγραμμα σε σχέση με τα προηγούμενα. Παρόλα αυτά, αν δώσουμε βάση στην πληθώρα των προβλημάτων του πρώτου μέρους θα αντιληφθούμε ότι δεν θα τα συναντήσουμε εδώ λόγω της διαφορετικής δομής του κώδικα (π.χ. προβλήματα με τους αλγόριθμους). Ακόμη έχουμε ήδη λύσει το θέμα της ανάγνωσης του αρχείου και της εκτύπωσης σε txt file.

Ας πάμε τώρα να αναφέρουμε αναλυτικότερα τα θέματα του προγράμματος. Πρώτα απ’ όλα δεν λειτουργούσε το inorder traversal και τυπώνονταν το αρχείο με την αρχική του μορφή. Σφάλμα το οποίο, διορθώθηκε καθώς ήταν ένα μηδαμινό λάθος στον κώδικα.

Επιστρέφουμε για μια στιγμή στο 1^{ο} μέρος. Εκεί είχαμε αποφασίσει να τυπώνουμε όλες τις περιπτώσεις μιας ημερομηνίας. Αυτή η επιλογή μας ήρθε να μας «τιμωρήσει» στο 2^{ο} μέρος. Πιο συγκεκριμένα, όταν κάνουμε αναζήτηση ενός στοιχείου εμφανίζονται όλες οι τιμές για την χ ημερομηνία. Αλλά, όταν κάνουμε τροποποίηση του στοιχείου αυτού επηρεάζεται πάντα το πρώτο instance και μόνο ενώ τα υπόλοιπα μένουν όπως ήταν πριν. Η διαγραφή αντιθέτως λειτουργεί κανονικά.

Αντιμετώπιση Προβλημάτων

Προχωράμε τώρα στο Value και στην εύρεση του μικρότερου και του μεγαλύτερου στο BST. Ερχόμαστε αντιμέτωποι με το θέμα ότι εμφανίζεται μόνο μία ημερομηνία

που αντιστοιχεί στο μικρότερο Value ή στο μεγαλύτερο παρόλο που αν ρίξουμε μια ματιά στο αρχείο αυτό δεν ισχύει. Λύση για αυτό δεν βρέθηκε, όμως επαληθεύεται ότι εντοπίζει σωστά τις τιμές.

Τέλος, ήρθε η ώρα να αναφερθούμε και στο hashing. Το search και εδώ λειτούργησε χωρίς προβλήματα. Το delete παρόλο που αν το τρέξουμε φαίνεται πως λειτουργεί, αν ξανακάνουμε search θα δούμε ότι δεν έχει γίνει η διαγραφή.

Συνοψίζοντας, η αλήθεια είναι πως δεν είχαμε πληθώρα αξιοσημείωτων προβλημάτων (τα μικρολαθάκια ήταν εύκολα αντιλήψιμα μετά από μια ματιά στον κώδικα ή με debugging), αλλά τα λάθη που αναφέρθηκαν παραπάνω μας ταλαιπώρησαν και το καθένα χρειάστηκε αρκετές ώρες ή και ημέρες για να λυθεί.