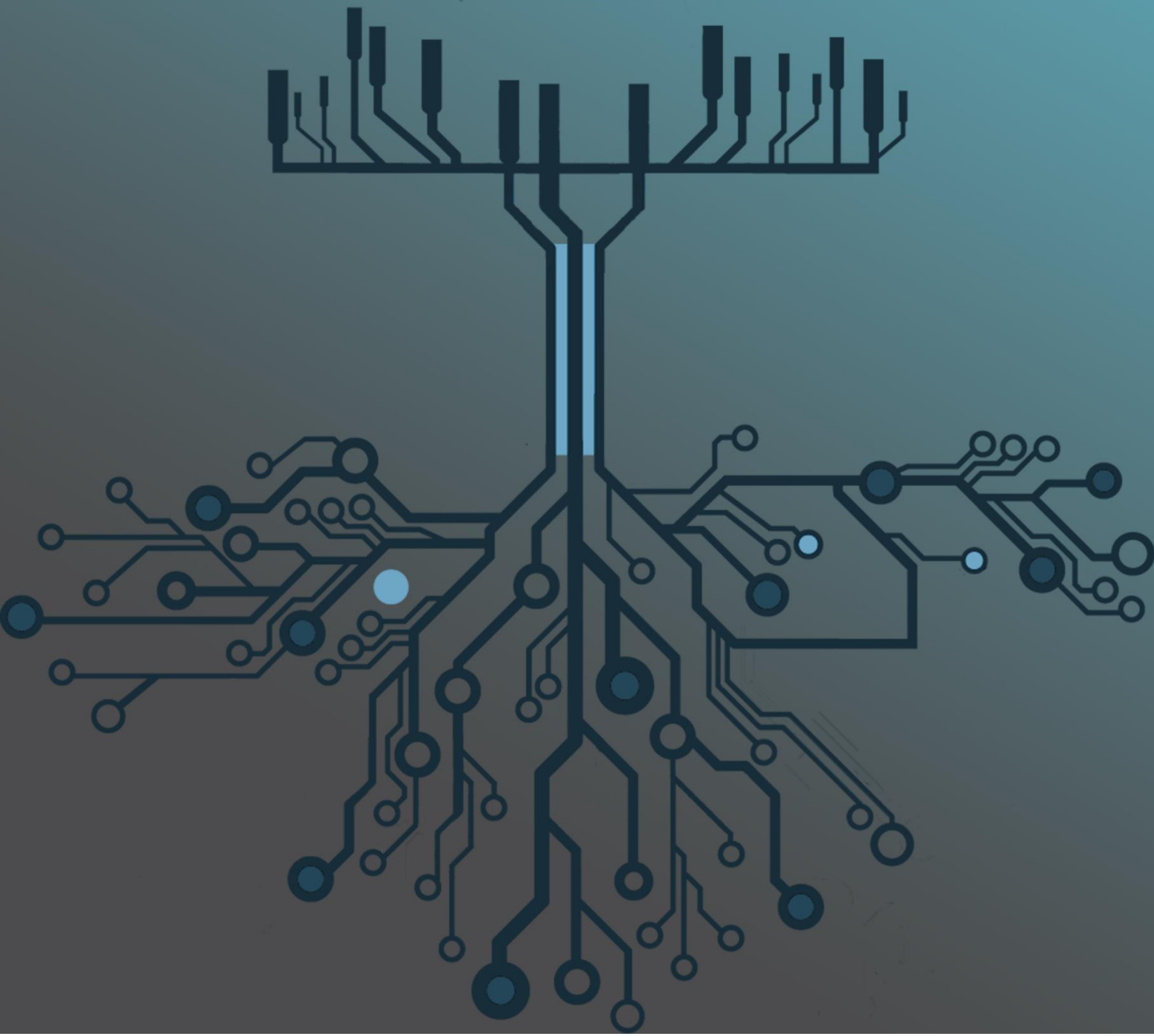
PROJECT: DATA STRUCTURES 2022

Creators

Ρουμπίνη - Μαρία Αγγουρά & Ιάσονας Παυλόπουλος



Μέλη Ομάδας

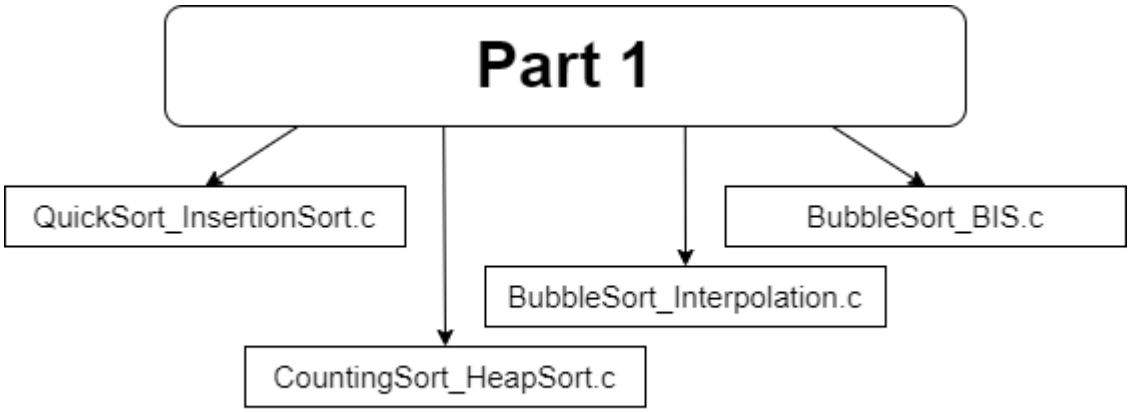
Όνομα: Παυλόπουλος Ιάσονας
ΑΜ: 1084565
E-mail: up1084565@upnet.gr

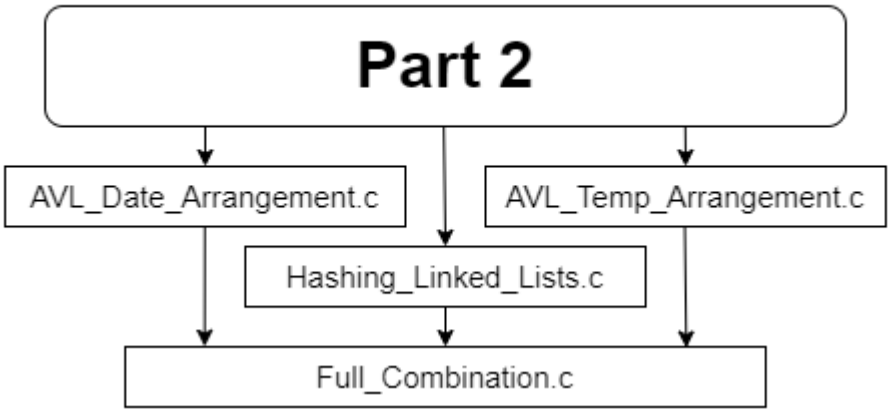
Όνομα: Αγγουρά Ρουμπίνη-Μαρία
ΑΜ: 1084634
E-mail: up1084634@upnet.gr

GitHub Repository: https://github.com/CallMeJasonYT/Data-Structures-And-Algorithms-Project-2022
Εδώ μπορείτε να βρείτε όλα τα αρχεία .C που σας έχουμε στείλει και στα Email σας. Σε περίπτωση που κάτι δεν λειτουργεί από το .rar μπορείτε να κατευθυνθείτε εδώ.

Διάγραμμα UML Map

Link (Σε περίπτωση που δεν έχει καλή ανάλυση η εικόνα): https://bit.ly/dsapMap





Επεξήγηση Κώδικα (Part 1)

QuickSort_InsertionSort.c

Αρχικά, για την υλοποίηση της ανάγνωσης του αρχείου ocean.csv και της εισαγωγής των δεδομένων από αυτό στο Struct “Ocean” χρησιμοποιήσαμε την “scanf” και με αυτόν τον τρόπο μπορέσαμε και χωρίσαμε και την ημερομηνία σε ξεχωριστές μεταβλητές Day/Month/Year. Ο συγκεκριμένος τρόπος είναι πιο απλός στην υλοποίηση, αλλά όπως θα δείτε και στα επόμενα κομμάτια του Project αλλάζουμε τον τρόπο που χειριζόμαστε τα δεδομένα. Χρησιμοποιώντας συναρτήσεις όπως η “FileCounter” και η “PrintArray”, μπορέσαμε να υπολογίσουμε πόσες γραμμές έχει το αρχείο δεδομένων και να εκτυπώσουμε τα δεδομένα του Struct που επιθυμούμε.

Στην συνέχεια υλοποιούμε τους αλγορίθμους Insertion Sort και Quick Sort. Ταξινομούμε τις ημερομηνίες με βάση τις τιμές της θερμοκρασίας. Ως γνωστόν, για τους δύο αυτούς αλγορίθμους ισχύουν οι παρακάτω χρόνοι:

Sorting Algorithm	Best Case	Average Case	Worst Case
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$

Επομένως είναι αναμενόμενο στις περισσότερες περιπτώσεις ο Quick Sort να ολοκληρώνεται πιο γρήγορα από τον Insertion Sort. Για να το επαληθεύσουμε αυτό, με την χρήση της βιβλιοθήκης <time.h> αρχικοποιήσαμε μία μεταβλητή τύπου “clock_t” στην αρχή του κάθε αλγορίθμου και στο τέλος του κάθε αλγορίθμου υπολογίσαμε την διαφορά των κύκλων ρολογιού που χρειάστηκαν ώστε να εκτελεσθεί ο κάθε αλγόριθμος ταξινόμησης. Έπειτα μετατρέψαμε τα Clocks σε ms και οδηγηθήκαμε στο εξής output για τους παραπάνω αλγορίθμους:

```
After Insertion Sort...

Time Elapsed: 0.003000
```

```
After Quick Sort...

Time Elapsed: 0.002000
```

Παρατηρούμε πως όντως είναι πιο γρήγορος ο Quick Sort, με χειρότερη περίπτωση $O(n^2)$, όπου n το πλήθος των στοιχειών του πίνακα. Και οι δύο αλγόριθμοι έχουν τον ίδιο χρόνο χειρότερης περίπτωσης όμως ο Quick Sort έχει πολύ καλύτερο αριθμό μέσης και καλύτερης περίπτωσης ($\theta(n\log(n))$ και $\Omega(n\log(n))$ αντίστοιχα).

Για την επαλήθευση της πραγματοποίησης των Sorting Algorithms χρησιμοποιήσαμε την συνάρτηση “PrintArray” όπως αναφέρθηκε και παραπάνω. Μπορείτε να την χρησιμοποιήσετε και εσείς για τον έλεγχο των τιμών του Struct.

CountingSort_HeapSort.c

Όπως και στο προηγούμενο αρχείο, χρησιμοποιούμε την “scanf” για την ανάγνωση και την εισαγωγή των δεδομένων στα Struct μας. Η κύρια διαφορά με το προηγούμενο πρόγραμμα είναι πως ο CountingSort δουλεύει μόνο για ακεραίους. Έτσι, δημιουργούμε ένα δεύτερο είδους Struct το Ocean2 το οποίο έχει ως ορίσματα τα day/month/year και την ακέραια τιμή Phosphate. Επίσης καθώς εισάγουμε τα δεδομένα στα Struct χρησιμοποιούμε την συνάρτηση της βιβλιοθήκης <math.h>, “round”, ώστε να στρογγυλοποιήσουμε την float τιμή phosphate από το αρχείο δεδομένων. Ομοίως με παραπάνω, χρησιμοποιώντας συναρτήσεις όπως η “FileCounter” και η “PrintArray”, μπορέσαμε να υπολογίσουμε πόσες γραμμές έχει το αρχείο δεδομένων και να εκτυπώσουμε τα δεδομένα του Struct που επιθυμούμε.

Στην συνέχεια υλοποιούμε τους αλγορίθμους Heap Sort και Counting Sort. Ταξινομούμε τις ημερομηνίες με βάση τις τιμές Phosphate. Για την υλοποίηση αυτών των αλγορίθμων ταξινόμησης δημιουργήσαμε τις συναρτήσεις SwapInt, SwapFloat, Find_Max και Heapify. Ως γνωστόν, για τους δύο αυτούς αλγορίθμους ισχύουν οι παρακάτω χρόνοι:

Sorting Algorithm	Best Case	Average Case	Worst Case
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Counting Sort	Ω(n +k)	θ(n +k)	O(n +k)

Επομένως είναι αναμενόμενο στις περισσότερες περιπτώσεις ο Counting Sort να ολοκληρώνεται πιο γρήγορα από τον Heap Sort. Για να το επαληθεύσουμε αυτό, με την χρήση της βιβλιοθήκης <time.h> αρχικοποιήσαμε μία μεταβλητή τύπου “clock_t” στην αρχή του κάθε αλγορίθμου και στο τέλος του κάθε αλγορίθμου υπολογίσαμε την διαφορά των κύκλων ρολογιού που χρειάστηκαν ώστε να εκτελεσθεί ο κάθε αλγόριθμος ταξινόμησης. Έπειτα μετατρέψαμε τα Clocks σε ms και οδηγηθήκαμε στο εξής output για τους παραπάνω αλγορίθμους:

```
After Heap Sort...

Time Elapsed: 0.001000
```

```
After Counting Sort...

Time Elapsed: 0.000000
```

Παρατηρούμε πως όντως είναι πιο γρήγορος ο Counting Sort με O(n+k) όπου n το πλήθος των στοιχειών του πίνακα και κ η μέγιστη τιμή από αυτές που ταξινομούμε. Ωστόσο, ο Counting Sort έχει το μειονέκτημα ότι λειτουργεί μόνο σε θετικούς ακέραιους με αποτέλεσμα να χάνουμε ένα σημαντικό μέρος της πληροφορίας μας, που σε κάποιες περιπτώσεις ίσως να ήταν χρήσιμο για την ακριβή ταξινόμηση των ημερομηνιών με βάση τις τιμές Phosphate.

Για την επαλήθευση της πραγματοποίησης των Sorting Algorithms χρησιμοποιήσαμε την συνάρτηση “PrintArray” όπως αναφέρθηκε και παραπάνω. Μπορείτε να την χρησιμοποιήσετε και εσείς για τον έλεγχο των τιμών του Struct.

BubbleSort_Interpolation.c

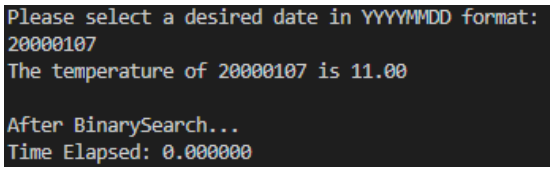
Σε αυτό το πρόγραμμα, χρειάστηκε να αλλάξουμε τον τρόπο με τον οποίο διαβάζουμε και εισάγουμε τα δεδομένα στα Struct. Επειδή, χρειάστηκε να υλοποιήσουμε ένα πρόγραμμα το οποίο να κάνει Binary Searching με όρισμα την Ημερομηνία, σημαίνει πως τα Δεδομένα μας έπρεπε να είναι ταξινομημένα ως προς την ημερομηνία. Έτσι, χρησιμοποιώντας τον buffer εισάγαμε τα δεδομένα μας στο Struct και αυτήν την φορά το Date το μετατρέψαμε από την μορφή MM/DD/YYYY σε ακέραιο της μορφής YYYYMMDD, ώστε να μπορέσουμε να ταξινομήσουμε τις ημερομηνίες κατά χρονολογική σειρά. Αυτό το πέτυχαμε με τις συναρτήσεις, RemoveChar και DateSwap, που υλοποιήσαμε.

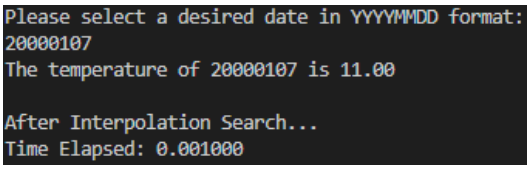
Έπειτα, με τον αλγόριθμο ταξινόμησης Bubblesort, ταξινομήσαμε τις ημερομηνίες με χρονολογική σειρά. Χρησιμοποιήσαμε τις συναρτήσεις SwapLi (Η ημερομηνία επειδή έχει 8 ψηφία χρειάζεται να είναι Long Int, εξού και το Swap Long Int) και Swapf ώστε να αλλάξουμε την σειρά της ημερομηνίας και της θερμοκρασίας αντίστοιχα.

Τέλος υλοποιήσαμε τους αλγορίθμους Binary Search και Interpolation Search. Ως γνωστόν, για τους δύο αυτούς αλγορίθμους αναζήτησης ισχύουν οι παρακάτω χρόνοι:

Searching Algorithm	Best Case	Average Case	Worst Case
Binary Search	Ω(1)	θ(log(n))	O(log(n))
Interpolation Search	Ω(1)	θ(log(log(n))	O(n)

Επομένως είναι αναμενόμενο στις περισσότερες περιπτώσεις ο Binary Search να ολοκληρώνεται πιο γρήγορα από τον Interpolation Search. Για να το επαληθεύσουμε αυτό, με την χρήση της βιβλιοθήκης <time.h> αρχικοποιήσαμε μία μεταβλητή τύπου “clock_t” στην αρχή του κάθε αλγορίθμου και στο τέλος του κάθε αλγορίθμου υπολογίσαμε την διαφορά των κύκλων ρολογιού που χρειάστηκαν ώστε να εκτελεσθεί ο κάθε αλγόριθμος ταξινόμησης. Έπειτα μετατρέψαμε τα Clocks σε ms και οδηγηθήκαμε στο εξής output για τους παραπάνω αλγορίθμους:





Παρατηρούμε πως όντως είναι πιο γρήγορος ο Binary Search με O(logn) όπου n το πλήθος των στοιχειών του πίνακα. Και οι δύο αλγόριθμοι έχουν τον ίδιο χρόνο καλύτερης περίπτωσης όμως ο Binary Search έχει καλύτερο αριθμό μέσης και πολύ καλύτερο χειρότερης περίπτωσης (θ(log(n)) και Ω(log(n)) αντίστοιχα).

Για την επαλήθευση της πραγματοποίησης των Sorting Algorithms χρησιμοποιήσαμε την συνάρτηση “PrintArray” όπως αναφέρθηκε και παραπάνω. Μπορείτε να την χρησιμοποιήσετε και εσείς για τον έλεγχο των τιμών του Struct.

Επεξήγηση Κώδικα (Part 2)

AVL_Date_Arrangement.c

Σε αυτό το πρόγραμμα χρησιμοποιούμε την ίδια μέθοδο για την ανάγνωση και την εισαγωγή των δεδομένων από το αρχείο ocean.csv. Ωστόσο, προσπαθούμε να φτιάξουμε ένα AVL δέντρο, επομένως πρέπει να δημιουργήσουμε ένα Struct (Node) με μεταβλητές Date, Temperature, Height και δύο pointers στον προηγούμενο και τους επόμενους κόμβους του (left, right).

Χρησιμοποιούμε μία σειρά συναρτήσεων που υλοποιήσαμε, ώστε να υλοποιήσουμε τις λειτουργίες Insertion, Deletion, Modification και Search στο AVL δέντρο μας. Αρχικά, Αρχικοποιούμε δυναμικά με την malloc το μέγεθος του AVL δέντρου και καθώς διαβάζουμε το αρχείο για κάθε εισαγωγή στοιχείου χρησιμοποιούμε την συνάρτηση Insertion, ώστε να εισάγουμε τα δεδομένα μας στο AVL δέντρο.

Εάν δεν υπάρχει τίποτα στο δέντρο, τότε χρησιμοποιούμε την συνάρτηση Node_Creation. Για την εξασφάλιση της δομής του AVL δέντρου, χρησιμοποιούμε τις συναρτήσεις Balancing, RightRotation, LeftRotation.
Έπειτα, αφότου το AVL δέντρο μας έχει σχηματιστεί, ο χρήστης έχει στην διάθεσή του ένα Menu που του επιτρέπει να Διαγράψει, Τροποποιήσει ή να Ψάξει μία τιμή Temperature στο AVL tree με βάση την ημερομηνία. Επίσης, δίνεται η δυνατότητα να εκτυπωθεί το δέντρο με ενδό-διατεταγμένη διάσχιση. Το menu απεικονίζεται παρακάτω:

```
Please Select One of the Following Options:
1) InOrder Traversal
2) Searching
3) Modification
4) Deletion
5) Exit
```

AVL_Temp_Arrangement.c

Όπως και στο παραπάνω πρόγραμμα, χρησιμοποιούμε τις συναρτήσεις Node_Creation, Insertion, RightRotation, LeftRotation και Balancing ώστε να χτίσουμε το AVL δέντρο μας. Ωστόσο, η ουσιαστική διαφορά είναι πως αντί να χρησιμοποιήσουμε την ημερομηνία για το διάβασμα του AVL δέντρου, να χρησιμοποιούμε την θερμοκρασία. Η διαφορά βρίσκεται στον κώδικα του Insertion όπως φαίνεται παρακάτω:

```
if(temp < node->temp){node->left = Insertion(node->left, date, temp);}
else if(temp > node->temp){node->right = Insertion(node->right, date, temp
else{return node;}

node->height = Height(node);
int balance = Balancing(node);

if (balance > 1 && temp < node->left->temp){return Right_Rotation(node);}
if (balance < -1  && temp > node->right->temp){return Left_Rotation(node);
if (balance > 1 && temp > node->left->temp)
{
    node->left = Left_Rotation(node->left);
    return Right_Rotation(node);
}
if (balance < -1 && temp < node->right->temp)
{
    node->right = Right_Rotation(node->right);
    return Left_Rotation(node);
}
return node;
```

```
//Performing the normal BST rotation
if(node == NULL){return Node_Creaction(date, temp);}
if(date < node->date){node->left = Insertion(node->left, date, temp);}
else if(date > node->date){node->right = Insertion(node->right, date, temp);}
else{return node;}

node->height = Height(node); //Updating the height of the node
int balance = Balancing(node); //Getting the balance factor of the node

//If this node becomes unbalanced, then there are 4 cases:
if (balance > 1 && date < node->left->date){return Right_Rotation(node);} //Left Left Case
if (balance <-1  && date > node->right->date){return Left_Rotation(node);} //Right Right Case
if (balance > 1 && date > node->left->date) //Left Right Case
{
    node->left = Left_Rotation(node->left);
    return Right_Rotation(node);
}
if (balance < -1 && date < node->right->date) //Right Left Case
{
    node->right = Right_Rotation(node->right);
    return Left_Rotation(node);
}
return node;
```

Τέλος, με τις συναρτήσεις Min_Temp και Max_Temp βρίσκουμε την Μέγιστη και την Ελάχιστη θερμοκρασία στο AVL Tree. Εφόσον το AVL δέντρο είναι ένα δυαδικό δέντρο αναζήτησης τότε τα δεδομένα αριστερού παιδιού < δεδομένα πατέρα < δεδομένα δεξιού παιδιού. Έτσι η ελάχιστη και η μέγιστη τιμή του όγκου θα βρίσκονται στο αριστερότερο και δεξιότερο φύλλο αντίστοιχα.

Hashing_Linked_Lists.c

Σε αυτήν την υλοποίηση, χρησιμοποιούμε Struct με μεταβλητές Date, Key, Temperature και pointer Next που δείχνει στο επόμενο στοιχείο σε περίπτωση που υπάρχει κατακερματισμός. Προκειμένου να ξεκινήσουμε την υλοποίηση με Linked Lists Hashing χρειαζόμαστε τη συνάρτηση Ascii_Sum για να προσθέσουμε τις Ascii τιμές των χαρακτήρων των ημερομηνιών από το αρχείο που είναι string της μορφής : “MM/DD/YYYY” προκειμένου να πάρουμε το κλειδί (key) της ημερομηνίας αυτής.

Έπειτα για την εισαγωγή της στον πίνακα χρησιμοποιούμε τη συνάρτηση κατακερματισμού h(k)=k mod m όπου m = Ο αριθμός γραμμών του αρχείου (όσοι δηλαδή και οι κόμβοι). Υλοποιούμε το hashing με αλυσίδες μέσω διασυνδεδεμένης λίστας (χρήση δεικτών) έτσι ώστε να αποφύγουμε τις συγκρούσεις (collisions). Όταν η θέση του πίνακα μίας ημερομηνίας με συγκεκριμένο κλειδί είναι ίδια με αυτήν ενός ήδη αποθηκευμένου κόμβου τότε έχουμε σύγκρουση και μέσω αλγορίθμου χειρισμού της λίστας προσθέτουμε το στοιχείο στο τέλος της.

Στον χρήστη δίνεται το εξής Menu:

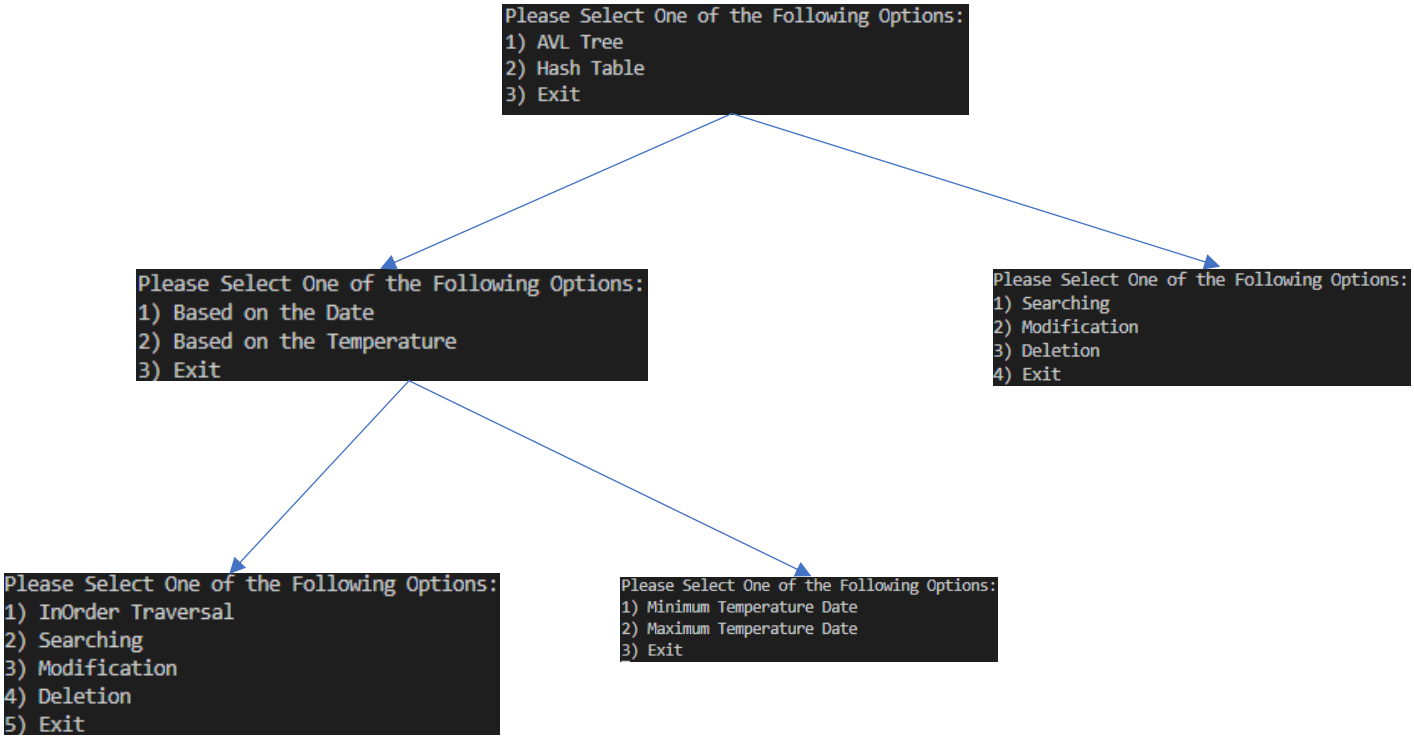
```
Please Select One of the Following Options:
1) Searching
2) Modification
3) Deletion
4) Exit
```

Για την αναζήτηση της θερμοκρασίας βάσει της ημερομηνίας που εισάγει ο χρήστης, αρχικά υπολογίζεται το κλειδί της και η θέση του πίνακα στην οποία αντιστοιχεί. Στη συνέχεια διατρέχουμε τη λίστα γραμμικά μέχρι το τέλος της για να το βρούμε.

Όμοια λειτουργεί και η συνάρτηση Modification για να αλλάξουμε την τιμή του Temperature. Η διαγραφή συγκεκριμένης ημερομηνίας πραγματοποιείται με σύνδεση του δείκτη next του προηγούμενου κόμβου από αυτόν που θέλουμε να διαγράψουμε με τον επόμενο κόμβο αυτού που θέλουμε να διαβάσουμε.

Full_Combination.c

Σε αυτό τον κώδικα, ενοποιήσαμε όλα τα κομμάτια κώδικα των 3 προηγούμενων αρχείων και δημιουργήσαμε ένα Menu ώστε ο χρήστης να μπορεί να επιλέξει ανάμεσα στο αν θέλει να υλοποιήσει ένα AVL δέντρο ή Hashing με Linked Lists. Έπειτα, εάν έχει επιλέξει AVL δέντρο, μπορεί να διαλέξει ανάμεσα σε η φόρτωση στο AVL θα γίνει με βάση την Ημερομηνία ή τη Θερμοκρασία ανά ημέρα.
Το Menu έχει την εξής μορφή:



Σας Ευχαριστούμε για τον Χρόνο σας