PROJECT: “DATA STRUCTURES 2021”

Δομές Δεδομένων

Δούρου Βασιλική-Ευαγγελία, AM:1072633 , έτος: 2o

Πεσκελίδης Παύλος, AM:1072483 , έτος: 2o

Μπελιμπασάκης Νικόλαος, AM:1072646 , έτος: 2o

Μπαλάφας Δημήτριος, AM:1072499 , έτος: 2o

Στο zip file παρατίθενται οι κώδικες του κάθε ερωτήματος.

**ΆΣΚΗΣΗ 1:**

Και για τους δύο αλγορίθμους ταξινόμησης, οι κώδικες εκτελέστηκαν στο Code::Blocks, στο DEV C/C++ και ήταν πλήρως λειτουργικοί.

Για τους 2 sort αλγόριθμους, στη main αρχικά, ανοίγουμε οποιοδήποτε από τα 3 αρχεία θέλουμε, μετράμε το πλήθος γραμμών που περιέχονται και δημιουργούμε ένα πίνακα με τον ίδιο αριθμό γραμμών, όπου θα αντιγραφούν τα περιεχόμενα του αρχείου. Έπειτα, μέσω της strtok() λαμβάνουμε τα στοιχεία που χρειαζόμαστε από κάθε γραμμή (στήλες date και open) και τα αποθηκεύουμε σε 2 πίνακες με αντίστοιχα ονόματα. Αμέσως μετά, καλούμε τις mergeSort και quicksort, μετράμε τους χρόνους και τέλος, καλούμε μία print συνάρτηση, για να δείξουμε τις 2 στήλες που εκλάβαμε μέσω των sort.

Για την εκτέλεση του Quick Sort, δημιουργούμε μία συνάρτηση η οποία παίρνει ως όρισμα το array που θέλουμε να ταξινομήσουμε, δηλαδή τον πίνακα arr, και το μέγεθος του.

void quicksort1(float arr[], int a, int b,char date[][15]){}

Επίσης έχουμε ως όρισμα και το date array που θα χρησιμοποιηθεί για να υπάρχει αντιστοίχηση μεταξύ των 2 στηλών. Η γενική ιδέα, είναι πως έχουμε ένα στοιχείο pivot που εδώ είναι το key, το οποίο είναι το πιο αριστερό στοιχείο στον πίνακα στη περίπτωση μας:

float key = arr[a];

Έτσι, κάνοντας συγκρίσεις στοιχείων αριστερά και δεξία με το pivot μέσω δύο increments (i,j) , τοποθετούμε τα στοιχεία μικρότερα από το κλειδί αριστερα από το ίδιο και όσα είναι μεγαλύτερα δεξιά. Μόλις τα conditions της quicksort ολοκληρωθούν, ξανατρέχουμε την Quick Sort στο αριστερό και στο δεξί μέλος ενώ ταυτόχρονα χρησιμοποιούμε την συνάρτηση swap για να αλλάξουμε τα στοιχεία του πίνακα date αντίστοιχα. To κομμάτι αυτό μπορούμε να το δούμε παρακάτω:

if (arr[a] > arr[i])

{

swap(arr, a, i,date);

quicksort1(arr, a, i - 1,date );

quicksort1(arr, i + 1, b,date);

} else

{

quicksort1(arr, a + 1, b,date);

}

Ακόμη, στη main καλούμε την νέα quicksort συνάρτηση για να είναι πιο εύκολο να τη καλέσουμε και να μη βάζουμε απευθείας τα ορίσματα:

void quicksort(float arr[], int len , char date[][15])

{

quicksort1(arr, 0, len - 1,date);

}

Για τη Merge Sort δημιουργούμε μία συνάρτηση void merge(float arr[], int l, int m, int r,char date[][15]) η οποία λαμβάνει τα δεδομένα του πίνακα arr[] και τα χωρίζουμε σε 2 πίνακες, τους L[] και R[]. Aυτό, συμβαίνει καθώς ο αλγόριθμος είναι divide and conquer, άρα διαιρεί τον πίνακα σε 2 υποπίνακες (γεγονός το οποίο θα γίνεται αναδρομικά μέχρι να φτάσουμε σε πίνακες του ενός στοιχείου) και αργότερα κάνει merge τους προσωρινούς πίνακες πίσω σε ένα γενικό πίνακα. Τέλος, τα υπόλoιπα στοιχεία που έχουν απομείνει αντιγράφονται στο τέλος του πίνακα,ενώ ταυτόχρονα γίνονται ανάλογες κινήσεις για την αντιστοίχηση των ημερομηνιών με τη στήλη open, όπως βλέπουμε παρακάτω από το απόσπασμα κώδικα:

i =0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

strcpy (date[k] , dateL[i]);

i++;

}

else

{

arr[k] = R[j];

strcpy (date[k] , dateR[j]);

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

strcpy (date[k] , dateL[i] );

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

strcpy (date[k] , dateR[j] ) ;

j++;

k++;

}

}

Η άλλη συνάρτηση που χρειαζόμαστε είναι η void mergeSort(float arr[], int l, int r, char date[][15]), η οποία χρησιμοποιείται για την ταξινόμηση των στοιχείων των υποπινάκων. Έπειτα, κάνουμε αναδρομή στην συνάρτηση για να ταξινομήσουμε το πρώτο μισό και το δεύτερο μισό ξεχωριστά και τελικά καλούμε την merge για να συνδυάσουμε τους 2 υποπίνακες σε έναν ταξινομημένο γενικό πίνακα. Για να έχουμε mergeSort σε 2 υποπίνακες, χρησιμοποιούμε έναν δείκτη m που θα χωρίσει τον πίνακα. Το m υπολογίζεται από τον τύπο : int m = l + (r - l) / 2; , o οποίος είναι καλύτερος από εναν απλό τύπο m = (l + r) / 2 , καθώς αποκλείει τη περίπτωση υπερχείλισης για μεγάλα l και h

Εκτελώντας τον αλγόριθμο του QuickSort για καθένα από τα τρία αρχεία τρεις φορές καταλήξαμε στις ακόλουθες πειραματικές τιμές:

Για το ainv.us.txt: 0.001002 seconds, 0.001001 seconds, 0.001000 seconds.

Για το agn.us.txt: 0.001000 seconds, 0.001000 seconds, 0.001001 seconds.

Για το ale.us.txt: 0.001001 seconds, 0.001001 seconds, 0.001000 seconds.

Εκτελώντας τον αλγόριθμο του ΜergeSort για καθένα από τα τρία αρχεία τρεις φορές καταλήξαμε στις ακόλουθες πειραματικές τιμές:

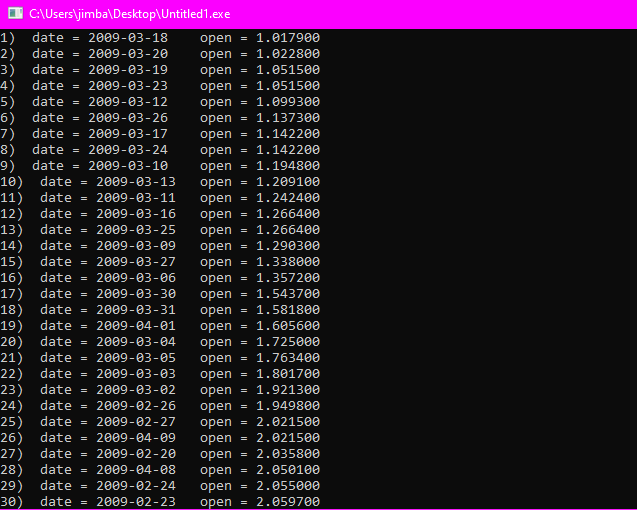
Για το ainv.us.txt: 0.001000 seconds, 0.001001 seconds, 0.001000 seconds.

Για το agn.us.txt: 0.001001 seconds, 0.001001 seconds, 0.001000 seconds.

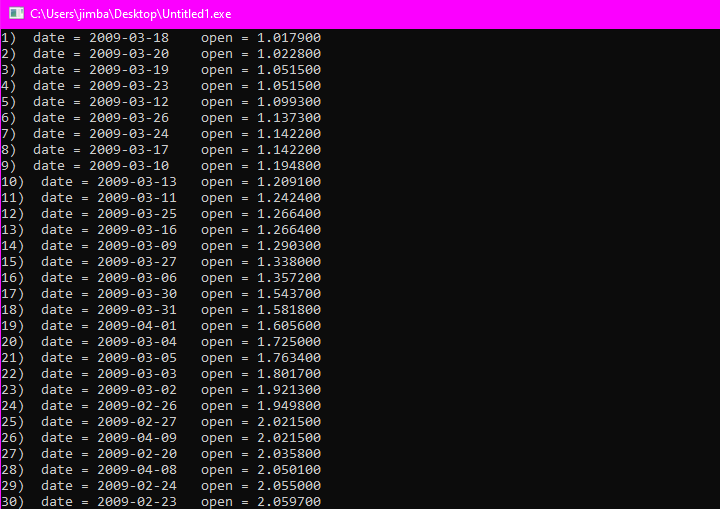
Για το ale.us.txt: 0.001001 seconds, 0.002007 seconds, 0.000999 seconds.

Παρατηρώντας τους χρόνους και για τους 2 αλγορίθμους, μπορούμε να δούμε πως οι χρόνοι είναι αρκετά κοντά και δε μπορεί από τα πειραματικά δεδομένα να φανερωθεί ο πιο γρήγορος με τα ίδια δεδομένα. Ωστόσο, ξέρουμε από τη θεωρία πως ο merge sort είναι πιο αποτελεσματικός και πιο γρήγορος από το quick sort σε μεγαλύτερους πίνακες, ενώ ο τελευταίος είναι γρηγορότερος σε μικρότερα μεγέθη πινάκων. Στη χειρότερη περίπτωση, μπορούμε να δούμε πως οι αλγόριθμοι διαφέρουν, αφού η Quick Sort έχει χρονική πολυπλοκότητα Ο(n2), ενώ η Merge Sort έχει Ο(nlogn) σε οποιαδήποτε περίπτωση.

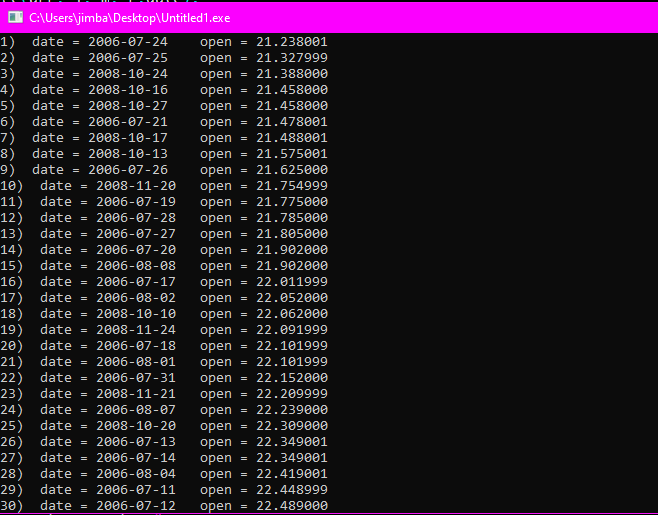
Στιγμιότυπο εκτέλεσης της Merge Sort για το αρχείο ainv.us.txt:



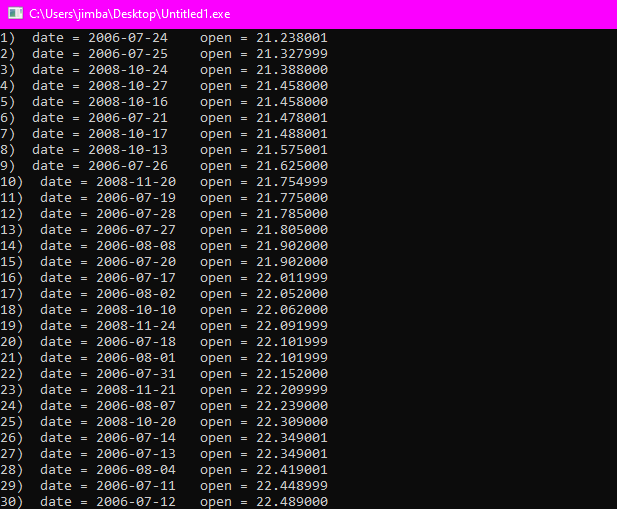
Στιγμιότυπο εκτέλεσης της Quick Sort για το αρχείο ainv.us.txt:



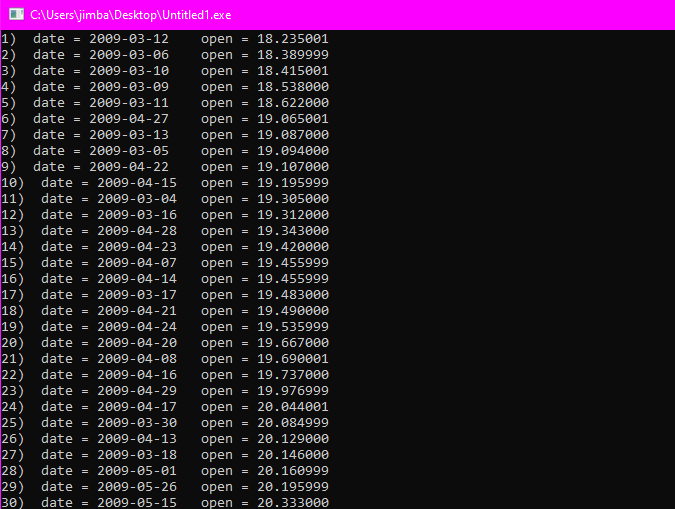
Στιγμιότυπο εκτέλεσης της Merge Sort για το αρχείο agn.us.txt:



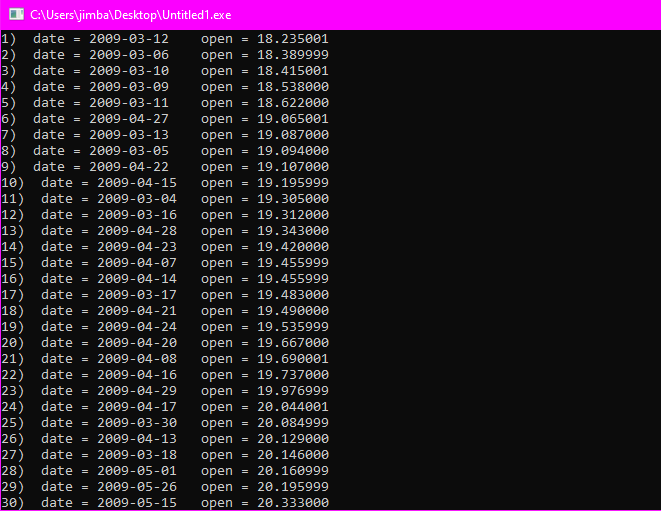
Στιγμιότυπο εκτέλεσης της Quick Sort για το αρχείο agn.us.txt:



Στιγμιότυπο εκτέλεσης της Merge Sort για το αρχείο ale.us.txt:



Στιγμιότυπο εκτέλεσης της Quick Sort για το αρχείο ale.us.txt:



**ΆΣΚΗΣΗ 2:**

Και για τους δύο αλγορίθμους ταξινόμησης, οι κώδικες εκτελέστηκαν στο Code::Blocks και ήταν πλήρως λειτουργικοί.

Στη main τόσο για τον Heap Sort όσο και για τον Counting Sort, αρχικά, ανοίγουμε ένα από τα τρία αρχεία που έχουν δοθεί, μετράμε το πλήθος των γραμμών που περιέχει και δημιουργούμε έναν πίνακα με τόσες γραμμές όσες αυτές που μετρήσαμε για να αντιγράψουμε τα περιεχόμενα του αρχείου, ώστε να τον διαπεράσουμε στη συνέχεια και να αποθηκεύσουμε σε έναν πίνακα δομών τις ημερομηνίες και τις τιμές κλεισίματος. Το πλήθος των γραμμών το μετράμε, καθώς, παρόλο που το κάθε αρχείο έχει διαφορετικό πλήθος στοιχείων, δεν θα είναι απαραίτητο να αλλάζουμε το μέγεθος των πινάκων ανάλογα με το ποιο αρχείο επιλέγουμε. Έπειτα, καλούμε την Heap Sort, ή την Counting Sort, υπολογίζουμε τον χρόνο εκτέλεσης που απαιτεί και, για να φαίνεται ότι ο αλγόριθμος μας έχει εκτελεστεί επιτυχώς, εκτελούμε μία printf για να δείξουμε τον ταξινομημένο, πλέον, πίνακα δομών μας.

Για την εκτέλεση του Heap Sort, έχουμε δημιουργήσει μία συνάρτηση η οποία παίρνει ως όρισμα το array που θέλουμε να ταξινομήσουμε, δηλαδή τον πίνακα δομών, και το μέγεθος του. Σύμφωνα με τη θεωρία, γνωρίζουμε ότι κάθε πίνακας S[1…n] είναι σωρός από τη θέση +1, οπότε ο κώδικας μας περιλαμβάνει δύο φάσεις, τη φάση δόμησης και τη φάση διαλογής. Η φάση δόμησης μετατρέπει τον υπόλοιπο μη διατεταγμένο πίνακα σε σωρό και η φάση διαλογής απομακρύνει το μεγαλύτερο στοιχείο, δηλαδή το στοιχείο που βρίσκεται στην πρώτη θέση του πίνακα.

Στον κώδικα μας η φάση δόμησης είναι η ακόλουθη:

if(l>0)

{

l=l-1;

j=l;

}

Ενώ η φάση διαλογής είναι η εξής:

else

{

temp=arr[r].close;

strcpy(d,arr[r].date);

arr[r].close=arr[0].close;

strcpy(arr[r].date,arr[0].date);

arr[0].close=temp;

strcpy(arr[0].date,d);

r=r-1;

j=0;

}

Στην πρώτη προσθέτουμε συνεχώς το στοιχείο l-1 όσο το l, δηλαδή το αριστερό μας άκρο, είναι μεγαλύτερο του μηδενός, ενώ στη δεύτερη, όταν πλέον έχουμε προσθέσει όλα τα στοιχεία στο σωρό, εναλλάσσουμε το στοιχείο arr[0], δηλαδή το μέγιστο στοιχείο του σωρού, με το arr[r] και μειώνουμε κατά ένα το r, δηλαδή το δεξί μας άκρο. Όσο το r, δηλαδή το δεξί άκρο, είναι μεγαλύτερο ή ίσο με 1, πέρα από τις δύο αυτές φάσεις, εκτελείται και ένας βρόγχος επανάληψης, στη διάρκεια του οποίου το στοιχείο arr[j] εναλλάσσεται με το μεγαλύτερο παιδί του, όπως φαίνεται παρακάτω:

s=arr[j].close;

strcpy(d,arr[j].date);

while(2\*j+1<=r)

{

k=2\*j+1;

if((k<r)&&(arr[k].close<arr[k+1].close))

{

k=k+1;

}

if(s<arr[k].close)

{

arr[j].close=arr[k].close;

strcpy(arr[j].date,arr[k].date);

j=k;

}

else

{

goto E;

}

}

E: arr[j].close=s; strcpy(arr[j].date,d);

Τέλος, καθώς μετά από αυτή την ταξινόμηση το αρχείο θα είναι ταξινομημένο μόνο ως προς τις τιμές κλεισίματος, προσθέτουμε έναν επιπλέον βρόγχο for, ο οποίος σε περίπτωση που βρει δύο ίσες τιμές κλεισίματος εναλλάσσει τις δύο ημερομηνίες ώστε η μικρότερη να εμφανίζεται πρώτη. Έτσι, το αρχείο ταξινομείται κατά αύξουσα σειρά πρώτα ως προς τις τιμές κλεισίματος και στη συνέχεια ως προς τις ημερομηνίες.

Για την εκτέλεση του Counting Sort, δημιουργούμε, αρχικά, μία συνάρτηση η οποία ονομάζεται calculateMax και υπολογίζει την μέγιστη τιμή κλεισίματος που έχει το αρχείο, στρογγυλοποιημένη προς τον πλησιέστερο ακέραιο.

Έπειτα, δημιουργούμε τη συνάρτηση Counting Sort, η οποία δέχεται ως όρισμα τον πίνακα που θέλουμε να ταξινομήσουμε, τον πίνακα της εξόδου, το μέγεθος του πίνακα και το μέγιστο στοιχείο του πίνακα που θέλουμε να ταξινομήσουμε. Πρώτα, αρχικοποιούμε τον βοηθητικό πίνακα C με μηδενικά και έπειτα αρχίζουμε να σαρώνουμε τον πίνακα arr από τα αριστερά προς τα δεξιά και σε εμφάνιση της στρογγυλοποιημένης τιμής arr[j] αυξάνουμε το C[arr[j]] κατά 1. Στη συνέχεια, σαρώνουμε τον πίνακα C από τα αριστερά προς τα δεξιά εκτελώντας την πράξη C[i]=C[i]+C[i-1] και έτσι σε κάθε θέση του πίνακα αυτού αποθηκεύεται το πλήθος των στοιχείων του πίνακα arr που είναι μικρότερα από αυτή. Τέλος, με τον ακόλουθο βρόγχο επανάληψης σαρώνουμε τον πίνακα arr από τα δεξιά προς τα αριστερά και αποθηκεύουμε την κάθε τιμή του στην κατάλληλη θέση του πίνακα εξόδου:

for(j=c;j>=1;j--)

{

B[C[(int)round(arr[j].close)]].close=round(arr[j].close);

strcpy(B[C[(int)round(arr[j].close)]].date,arr[j].date);

C[(int)round(arr[j].close)]=C[(int)round(arr[j].close)]-1;

}

Ο αλγόριθμος Counting Sort, σε αντίθεση με τον Heap Sort, δεν χρειάστηκε επιπλέον σάρωση για να γίνει η κατάταξη και με βάση τις ημερομηνίες, καθώς κάνοντας στο τελευταίο βήμα τη σάρωση από τα δεξιά προς τα αριστερά, τα στοιχεία περνάνε στον τελικό πίνακα με αύξουσα σειρά.

Εκτελώντας τον κώδικα του Heap Sort για καθένα από τα τρία αρχεία τρεις φορές καταλήξαμε στις ακόλουθες πειραματικές τιμές:

Για το ainv.us.txt: 0.000990sec, 0.001993sec, 0.000996sec.

Για το ale.us.txt: 0.001994sec, 0.000976sec, 0.001995sec.

Για το agn.us.txt: 0.002990sec, 0.002972sec, 0.003989sec.

Αντίστοιχα, εκτελώντας τον κώδικα του Counting Sort για καθένα από τα τρία αρχεία τρεις φορές καταλήξαμε στις ακόλουθες πειραματικές τιμές:

Για το ainv.us.txt: 0.000998sec, 0.000997sec, 0.000998sec.

Για το ale.us.txt: 0.000996sec, 0.000981sec, 0.000999sec.

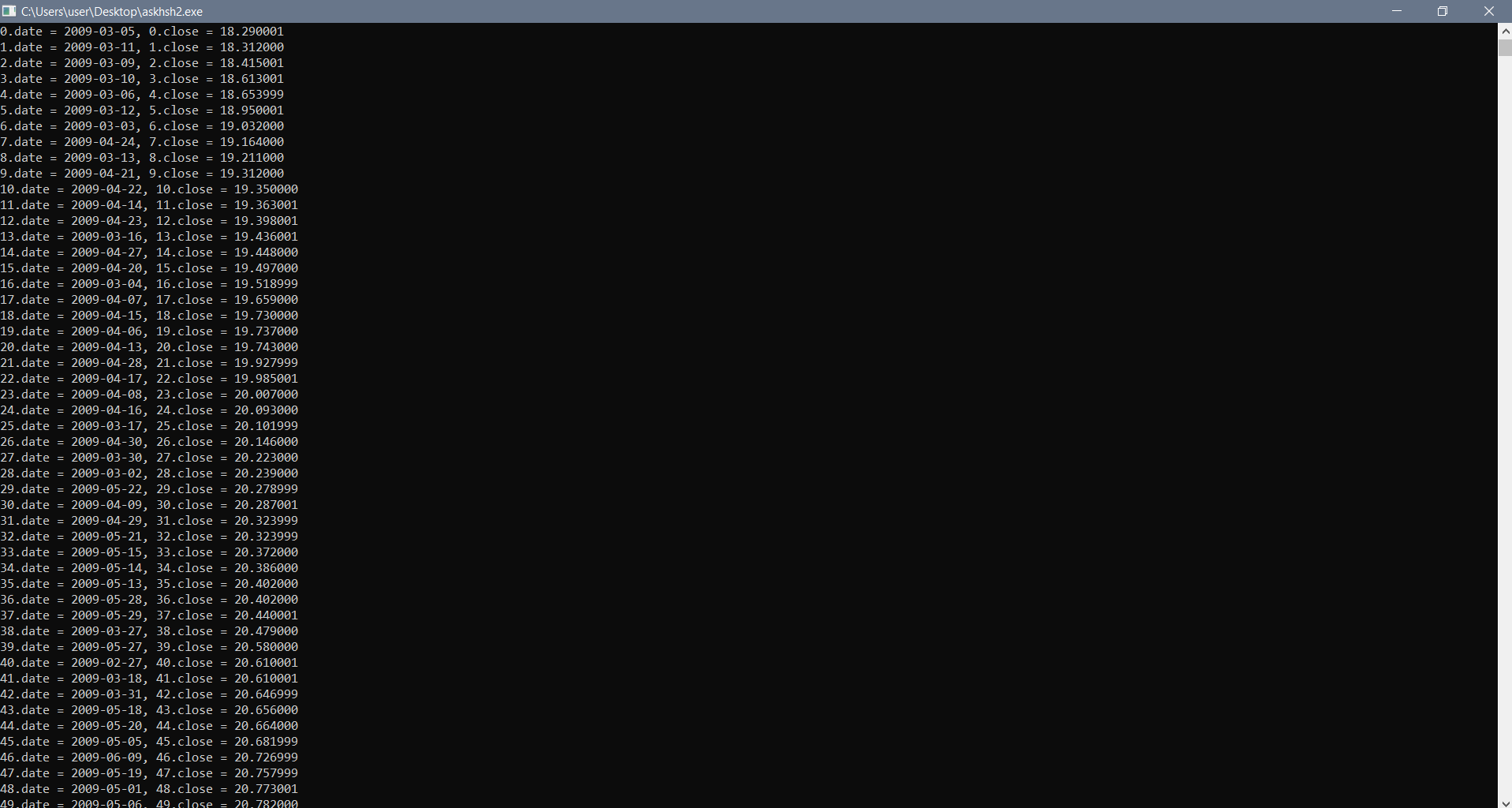
Για το agn.us.txt: 0.000999sec, 0.000998sec, 0.001007sec.

Παρατηρούμε ότι και για τα τρία αρχεία οι χρόνοι εκτέλεσης του Counting Sort είναι πιο γρήγοροι από του χρόνους εκτέλεσης του Heap Sort.

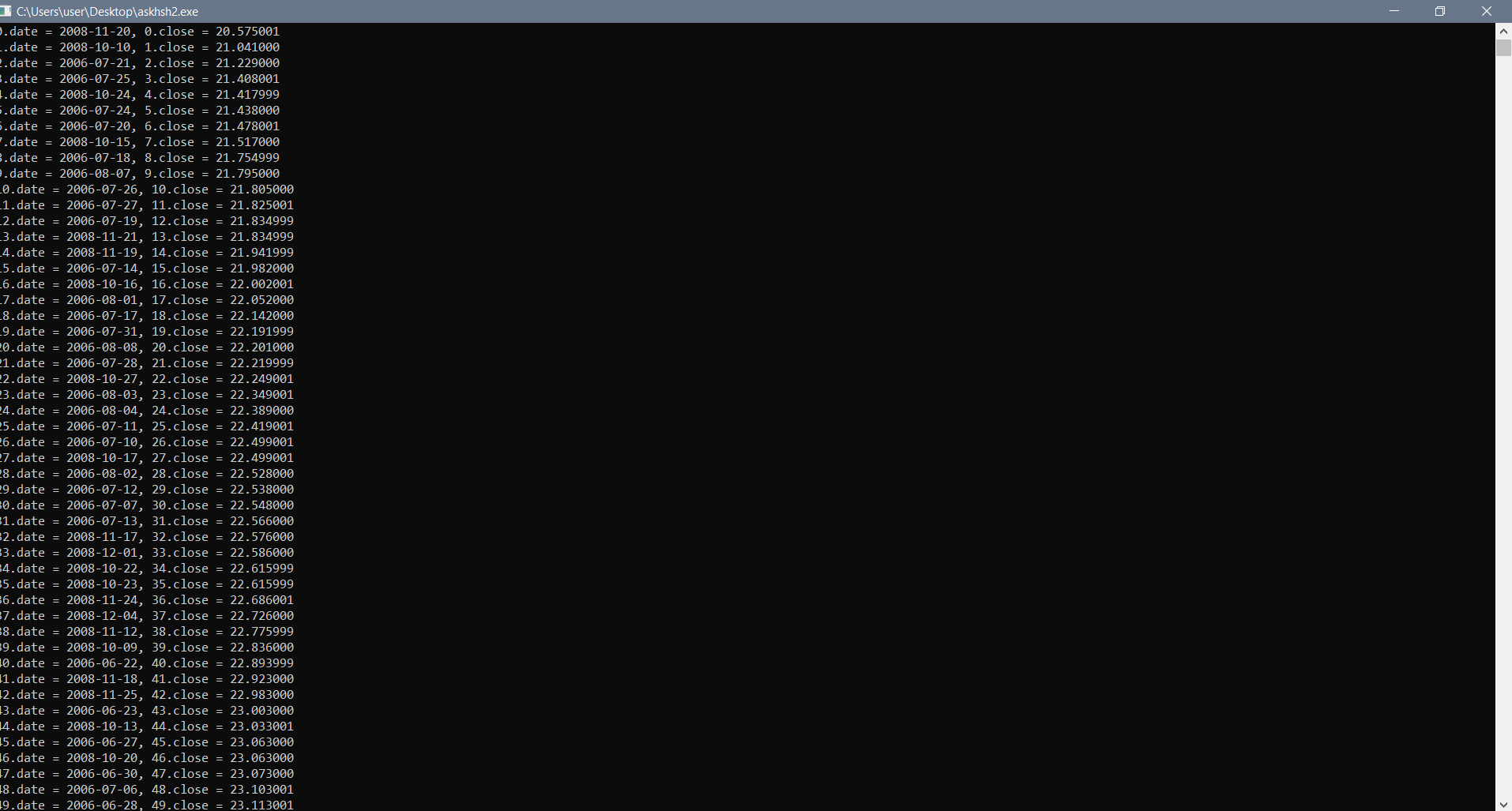
Όπως είναι γνωστό από τη θεωρία, το συνολικό κόστος του Counting Sort είναι O(n+k), όπου k ο μέγιστος ακέραιος που μπορεί να έχει ως τιμή close το αρχείο μας και n ο αριθμός των στοιχείων. Καθώς και στα τρία αρχεία που δόθηκαν το k ήταν πολύ μικρότερο από το n, ο αλγόριθμος του Counting Sort αποκτάει καλύτερους χρόνους από τους αλγόριθμους που βασίζονται σε συγκρίσεις, όπως ο Heap Sort, ο οποίος έχει χρονική πολυπλοκότητα O(nlogn).

Μερικά στιγμιότυπα από την εκτέλεση του Heap Sort είναι τα ακόλουθα (πρώτα στοιχεία μετά την ταξινόμηση):

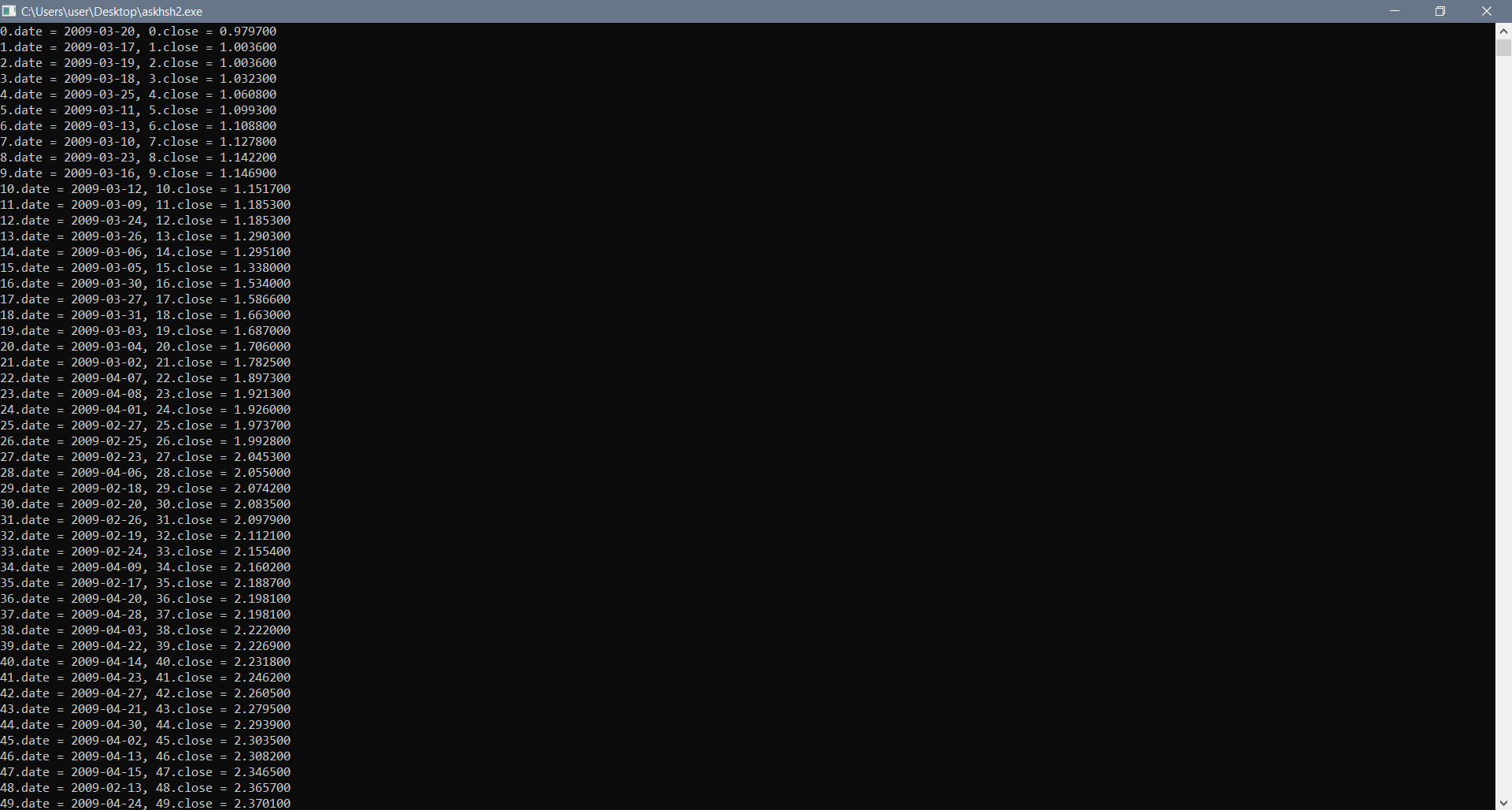
Για το αρχείο ale.us.txt:



Για το αρχείο agn.us.txt:

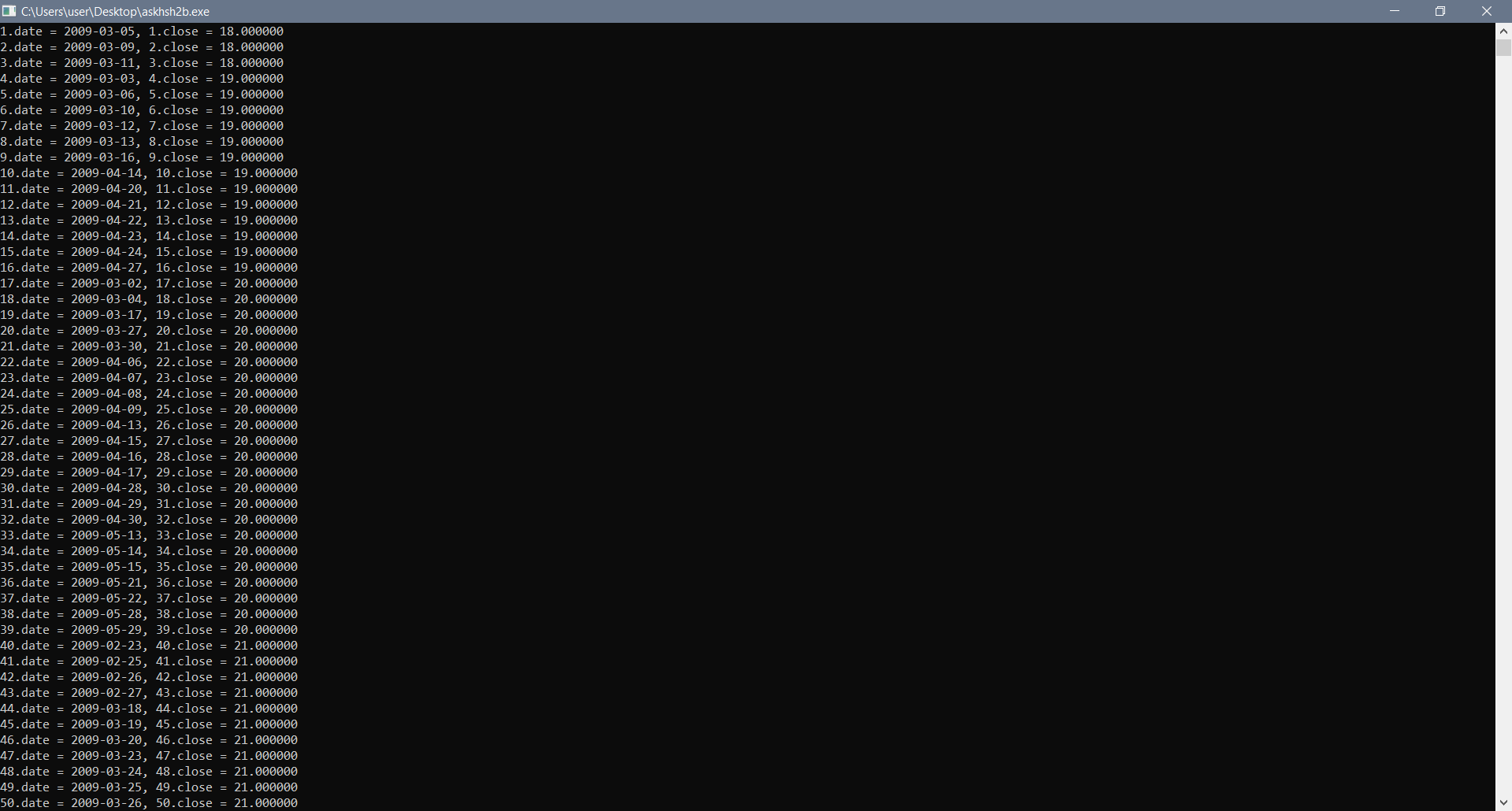


Για το αρχείο ainv.us.txt:

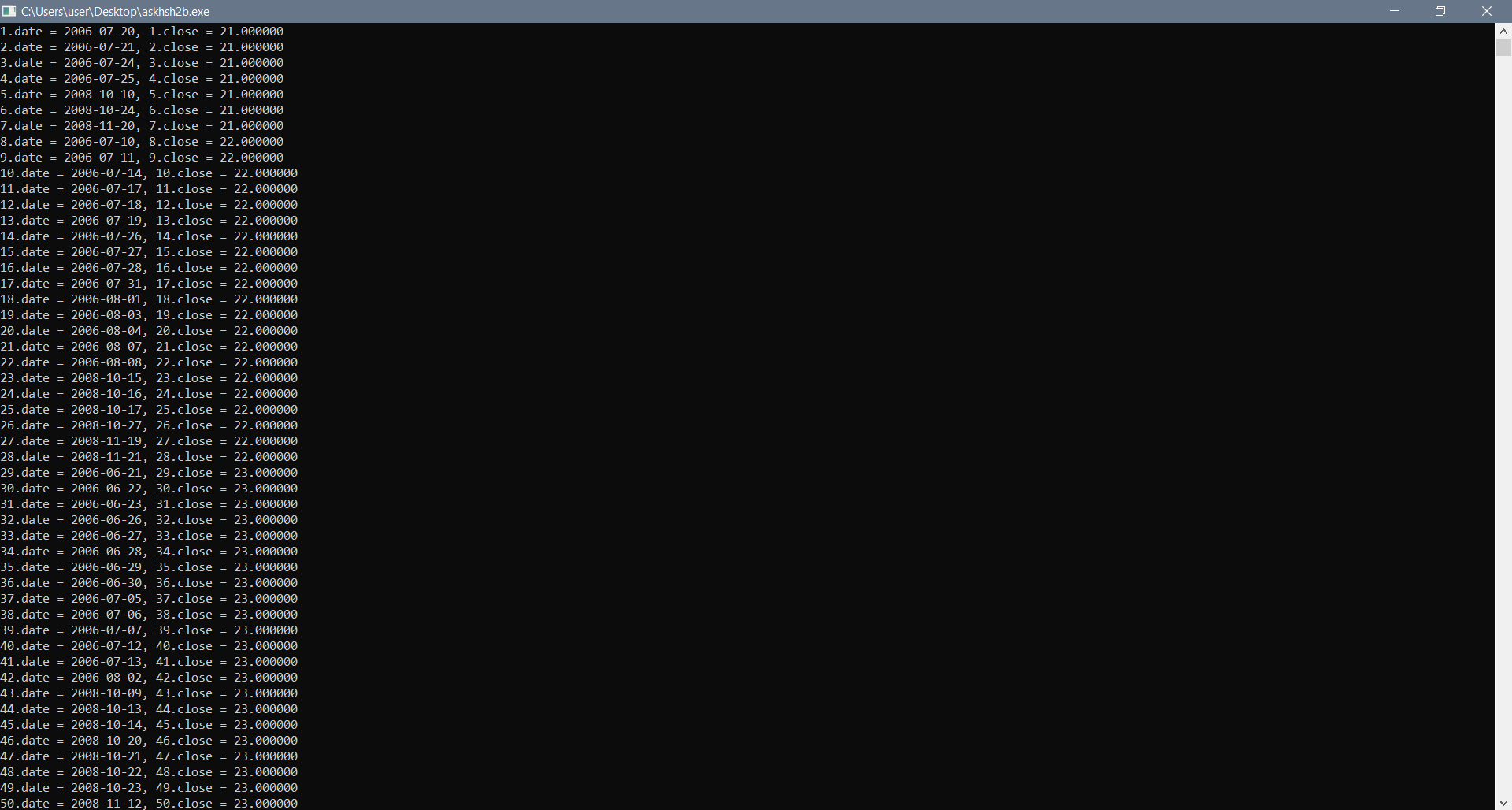


Αντίστοιχα, μερικά στιγμιότυπα από την εκτέλεση του Counting Sort είναι τα ακόλουθα (πρώτα στοιχεία μετά την ταξινόμηση):

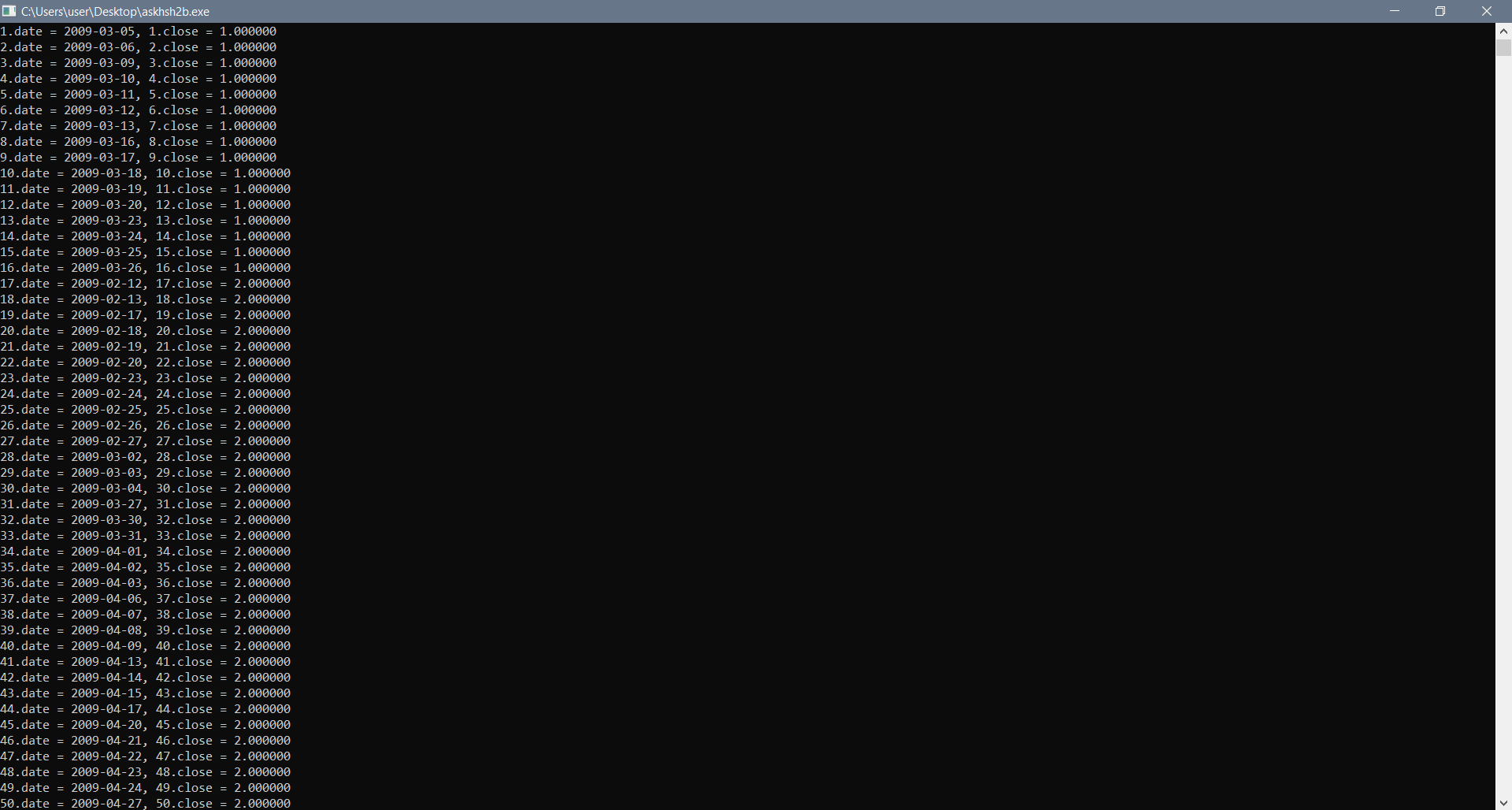
Για το αρχείο ale.us.txt:



Για το αρχείο agn.us.txt:



Για το αρχείο ainv.us.txt:



**ΆΣΚΗΣΗ 3:**

Και για τους δύο αλγορίθμους ταξινόμησης, οι κώδικες εκτελέστηκαν στο Code::Blocks, στο Dev C++ και για όσα τυχαία στιγμιότυπα ελέγχτηκαν ήταν πλήρως λειτουργικοί.

Στις 2 main, τόσο στη Binary Search , όσο και στην Interpolation Search ανοίγουμε ένα από τα τρία αρχεία, οποιοδήποτε θέλουμε και όπως στα προηγούμενα ερωτήματα αντιγράφουμε τα περιεχόμενα κάθε γραμμής σε έναν πίνακα, κάνοντας extract δύο συγκεκριμένες στήλες, στη περίπτωση μας τις date και volume. Έπειτα, ζητάμε από τον χρήστη που εκτελεί το πρόγραμμα να δώσει μία ημερομηνία σε συγκεκριμένο format (YYYY-MM-DD) και αν δε βρεθεί (αυτό ελέγχεται με μία μεταβλητή found), εμφανίζεται ανάλογο μήνυμα, για να αποφευχθούν οι λάθος είσοδοι. Tέλος, καλώντας την κάθε συνάρτηση που βρίσκεται ανάμεσα από τις gettimeofday() εμφανίζεται ο αντίστοιχος αριθμός όγκου (η printf υπάρχει μέσα στη συνάρτηση).

Για τον αλγόριθμο Binary Search, καλούμε τη συνάρτηση

void binarySearch(char arr[][15], int left, int right, char sr[],int volume[])

με ορίσματα τους πίνακες που θα χρειαστούμε και τα όρια του πίνακα που θα ψάξουμε, δηλαδή 1 η αρχική θέση, καθώς από εκεί αρχίζουν τα δεδομένα μας, έως τη τελευταία θέση -1. Έπειτα, ορίζουμε το μεσαίο στοιχείο ως

int middle = left + (right - left) / 2;

και ελέγχουμε αν το ζητούμενο στοιχείο είναι το μεσαίο. Αν το στοιχείο, είναι μικρότερο από το m, τότε κάνουμε αναδρομικό Search στο κομμάτι αριστερά από το m με την

binarySearch(arr, left, middle - 1, sr,volume); ,

ενώ αν είναι μεγαλύτερο από το m, τότε κάνουμε αναδρομικό Search στο κομμάτι δεξιά από το m με την

binarySearch(arr, middle + 1, right, sr,volume); .

Τέλος, η print που εμφανίζει τον ζητούμενο όγκο, καλείται μέσα από την συνάρτηση binarySearch στο εξής κομμάτι κώδικα:

int middle = left + (right - left) / 2;

int val = strcmp(arr[middle],sr);

if (val==0)

{

found = 1 ;

printf("Volume : %d", volume[middle]);

}

Bλέπουμε, ότι πρώτα ορίζουμε το μεσαίο στοιχείο, το οποίο είναι int άρα δεν ανησυχούμε για δεκαδικούς που μπορεί να προκύψουν. Επιπρόσθετα, ορίζουμε μία μεταβλητή val που συγκρίνει το ζητούμενο με το μεσαίο στοιχείο του πίνακα, αν δε ταυτίζονται, η val δίνει σήμα να εκτελεστεί αριστερά ή δεξιά από το μεσαίο στοιχείο η binarySearch. Με αυτό το τρόπο, γίνονται συνεχώς συγκρίσεις μέχρι να βρεθούμε σε ένα μέγεθος πίνακα όπου το middle θα ταυτίζεται με το ζητούμενο, γεγονός το οποίο στη χειρότερη περίπτωση θα καλέσει την τόσες φορές μέχρι το μοναδικό στοιχείο στον υποπίνακα να είναι το middle, όπου θα εκπληρωθεί η

found = 1;

Για τον αλγόριθμο Interpolation Search, καλούμε την συνάρτηση:

void interpolationSearch(int arr[], int low, int high, int z,int volume[])

η οποία έχει ως όρισμα τον πίνακα στον οποίο θα γίνει η αναζήτηση, το μέγεθος του πίνακα και την ημερομηνία που θα αναζητήσει ο χρήστης. Αρχικά, βάζουμε ως άκρα low και high τις τιμές 1 και το μέγεθος του πίνακα μειωμένο κατά 1. Επιπρόσθετα, υπολογίζουμε τη μεταβλητή thesi, η οποία χρησιμοποιείται από τον αλγόριθμο, καθώς αντίθετα με το binary search, o interpolation search μεταφέρει το κλειδί ανάλογα με το κλειδί που ψάχνουμε σύμφωνα με τη formula της μεταβλητής thesi:

thesi = low + (((double)(high - low) / (arr[high] - arr[low])) \* (z - arr[low]));

Ο τύπος δομείται ως το άθροισμα του αριθμού στη πρώτη θέση (low) συν το κλάσμα με αριθμητή το τελευταίο στοιχείο μείον το πρώτο στοιχείο σε double format δια τον αριθμό στον πίνακα arr με θέση τον αριθμό high μείον τον αριθμό στον πίνακα arr με θέση τον αριθμό low επί τον αριθμό που ζητάμε μείον τον αριθμό στον πίνακα arr με θέση τον αριθμό low.

Αν η θέση του πίνακα με τιμή τη μεταβλητή thesi είναι το στοιχείο που ζητάμε, τότε επιστρέφεται ο αντίστοιχος όγκος. Αν το στοιχείο που ζητάμε είναι μεγαλύτερο από αυτό στη θέση arr[thesi], τότε εκτελούμε αναδρομικά την interpolationSearch στο δεξιό υποπίνακα.Ενώ, αν το στοιχείο που ζητάμε είναι μικρότερο από αυτό στη θέση arr[thesi], τότε εκτελούμε αναδρομικά την interpolationSearch στον αριστερό υποπίνακα.Τα παραπάνω είναι εμφανή στο κομμάτι κώδικα:

int thesi;

if (low <= high && z >= arr[low] && z <= arr[high])

{

thesi = low + (((double)(high - low) / (arr[high] - arr[low])) \* (z - arr[low]));

if (arr[thesi] == z)

{

found = 1 ;

printf("Volume: %d", volume[thesi]);

}

if (arr[thesi] <z)

return interpolationSearch(arr, thesi + 1, high, z,volume);

if (arr[thesi] >z)

return interpolationSearch(arr, low, thesi - 1, z,volume);

}

Τέλος, στη main() χρησιμοποιούμε ένα συγκεκριμένο κομμάτι κώδικα για να λαμβάνουμε το date με την scanf. Mε αυτό το τρόπο, χρησιμοποιούμε τις συναρτήσεις strcpy() και strcat() για να δημιουργήσουμε ένα string από τη δοθείσα ημερομηνία, το οποίο μπορούμε να χρησιμοποιήσουμε για να βρούμε το Volume. Αυτό το κομμάτι κώδικα παρατίθεται παρακάτω:

da = strtok(date1,s1);

strcpy(year,da); // pairnei ton xrono

da = strtok(NULL,s1);

strcpy(month,da); // pairnei ton mhna

da = strtok(NULL," ");

strcpy(day,da); // pairnei thn mera

strcpy(date1,year);

strcat(date1,month);

strcat(date1,day);

int dateI = atoi(date1);

Εκτελώντας τη Binary Search, έχουμε τους εξής χρόνους εκτέλεσης για 10 διαφορετικές εισόδους στο agn.us.txt:

0.000999sec,0.000979sec,0.001004sec,0.001003sec,0.001015sec,0.001002sec,0.001000sec,0.000999sec,0.000998sec,0.001002sec. Άρα ο μέσος χρόνος περίπτωσης, είναι: 0.0010001sec

Εκτελώντας τη Binary Search, έχουμε τους εξής χρόνους εκτέλεσης για 10 διαφορετικές εισόδους στο ainv.us.txt:

0.001001sec,0.001002sec,0.001000sec,0.001001sec,0.000999sec,0.001000sec,0.001001sec,0.001003sec,0.001001sec,0.001000sec. Άρα ο μέσος χρόνος περίπτωσης, είναι: 0,0010008sec

Εκτελώντας τη Binary Search, έχουμε τους εξής χρόνους εκτέλεσης για 10 διαφορετικές εισόδους στο ale.us.txt:

0.000533sec,0.001000sec,0.001002sec,0.001000sec,0.001000sec,0.001002sec,0.001001sec,0.001003sec,0.001000sec,0.001001sec. Άρα ο μέσος χρόνος περίπτωσης, είναι: 0,0009542sec

Εκτελώντας την Interpolation Search, έχουμε τους εξής χρόνους εκτέλεσης για 10 διαφορετικές εισόδους στο agn.us.txt:

0.001003sec,0.000975sec,0.001003sec,0.001001sec,0.001021sec,0.001005sec,0.001000sec,0.001000sec,0.001120sec,0.001004sec. Άρα ο μέσος χρόνος περίπτωσης, είναι: 0,0010132sec

Εκτελώντας την Interpolation Search, έχουμε τους εξής χρόνους εκτέλεσης για 10 διαφορετικές εισόδους στο ainv.us.txt:

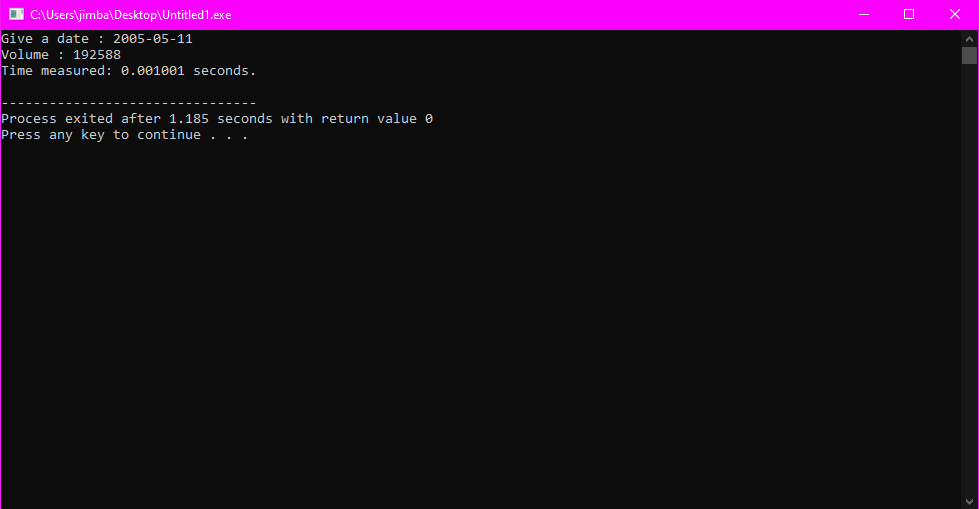
0.001000sec,0.001000sec,0.001002sec,0.001001sec,0.001001sec,0.000999sec,0.001001sec,0.001004sec,0.001001sec,0.001001sec. Άρα ο μέσος χρόνος περίπτωσης,είναι: 0,001001sec

Εκτελώντας την Interpolation Search, έχουμε τους εξής χρόνους εκτέλεσης για 10 διαφορετικές εισόδους στο ale.us.txt:

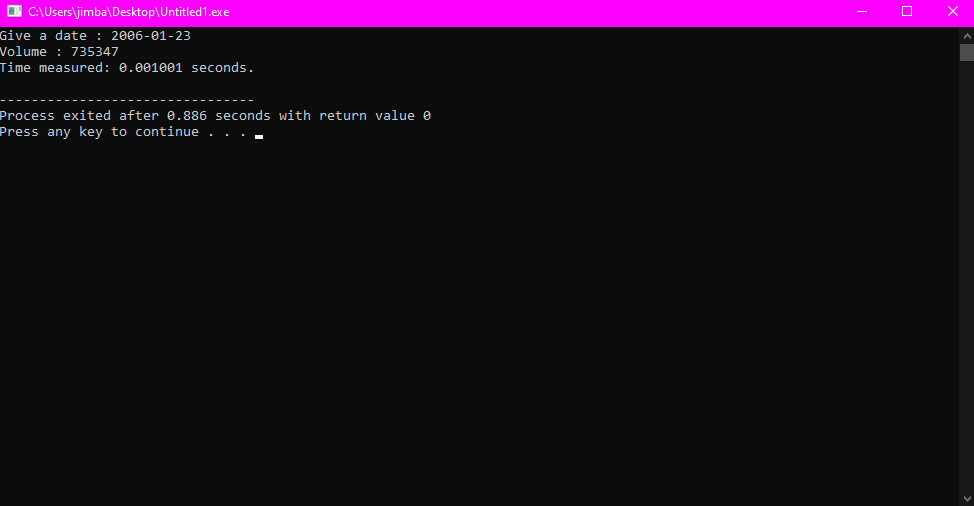
0.001003sec,0,001000sec,0.000728sec,0.001002sec,0.001002sec,0.001003sec,0.001002sec,0.000679sec,0.001002sec,0.001001sec. Άρα ο μέσος χρόνος περίπτωσης, είναι: 0,0009422sec

Οι χρόνοι μέσης περίπτωσης για τους 2 αλγόριθμους δείχνουν να είναι αρκετά κοντά και γι’ αυτό το λόγο μπορούμε να καταφύγουμε στη θεωρία για να καταλήξουμε σε ένα συμπέρασμα. Η μέση περίπτωση ενός επιτυχημένου search για την binary search είναι Ο(logn), ενώ η μέση περίπτωση ενός επιτυχημένου search για interpolation search είναι O(log(logn)). Γνωρίζουμε, πως το Ο(logn) είναι γρηγορότερο από O(log(logn)) μετά από ένα συγκεκριμένο όριο, ωστόσο γενικά σε μέση περίπτωση, η χρονική πολυπλοκότητα O(log(logn)) είναι καλύτερη και ταυτόχρονα γνωρίζουμε πως σε γενικά πλαίσια η Interpolation search είναι μία καλύτερη εκδοχή της Binary Search. Όσον αφορά τη κατανομή του Data Set, η Binary Search σε μικρότερες εισόδους, οι οποίες άκολουθούν ομοιόμορφη κατανομή, είναι αρκετά καλή, καθώς υπάρχει ισάξια πιθανότητα το στοιχείο που θέλουμε να βρίσκεται στο middle(Μπορούμε να δούμε και τον αλγόριθμο [uniform binary search](https://en.wikipedia.org/wiki/Uniform_binary_search) που είναι βελτιωμένος αλγόριθμος για ομοιόμορφες κατανομές). Ωστόσο, αν δεν είναι ομοιόμορφη η κατανομή μπορούμε να καταφύγουμε στη χειρότερη περίπτωση, όπου χρονική πολυπλοκότητα είναι O(logn). Δεν υπάρχουν πολλά στοιχεία για μη ομοιόμορφες κατανομές για τη binary search , ωστόσο υπάρχουν αρκετά θετικά χαρακτηριστικά που αναφέρονται για την Interpolation search, όσο έχει να κάνει με τις κατανομές. Η [interpolation search](https://en.wikipedia.org/wiki/Interpolation_search) , προτιμάται όταν έχουμε να κάνουμε με ομοιόμορφες κατανομές, καθώς η χρονική πολυπλοκότητα μετατρέπεται από O(n) για μέγεθος n σε Ο(log logn). H interpolation search υπερέχει σε αυτή τη περίπτωση, διότι για να υπολογιστεί το key ή αλλιώς probe, χρησιμοποιούμε έναν ιδιαίτερο τύπο που είναι καλύτερος από το να χρησιμοποιούμε το μεσαίο στοιχείο για κλειδί.

Στιγμιότυπο για την εκτέλεση της Interpolation Search στο αρχείο ale.us.txt:



Στιγμιότυπο για την εκτέλεση της Binary Search στο αρχείο agn.us.txt:



**ΆΣΚΗΣΗ 4:**

Και για τους δύο αλγορίθμους ταξινόμησης, οι κώδικες εκτελέστηκαν στο Code::Blocks και για όσα τυχαία στιγμιότυπα ελέγχτηκαν ήταν πλήρως λειτουργικοί.

Στη main τόσο για το Binary Interpolation Search όσο και για τη βελτιωμένη εκδοχή του, αρχικά, ανοίγουμε ένα από τα τρία αρχεία που έχουν δοθεί, όπως και στο προηγούμενο ερώτημα, μετράμε τις γραμμές του και αντιγράφουμε τα περιεχόμενα του σε ένα πίνακα δομών, με στοιχεία την ημερομηνία και τον όγκο. Στη συνέχεια, ζητάμε από τον χρήστη να μας δώσει μία ημερομηνία για την οποία θέλει να μάθει τον όγκο συναλλαγών και την παίρνουμε με μία scanf. Σε περίπτωση που η ημερομηνία δεν έχει παύλα (-) για να διακρίνει το έτος από το μήνα και την ημέρα, με τον ακόλουθο βρόγχο επανάληψης while, ζητάμε από τον χρήστη να μας ξαναδώσει μία ημερομηνία στη σωστή μορφή:

while((strchr(givenDate,'-'))==NULL)

{

printf("Wrong form. Date must be in the form of year-month-date.\n");

printf("Give the date you wish to check(our records are in the period of 2005-2017):");

scanf("%s", givenDate);

}

Επίσης, αφού παρατηρήσαμε ότι όλα τα αρχεία περιέχουν πληροφορίες για τα έτη 2005 έως 2017, με τον ακόλουθο βρόγχο επανάληψης while ζητάμε να μας δοθεί μία ημερομηνία σε αυτό το διάστημα:

while((givendate<=20041231)||(givendate>=20180101))

{

printf("Our records do not cover this date.\n");

printf("Give the date you wish to check(our records are in the period of 2005-2017):");

scanf("%s", givenDate);

sscanf(givenDate,"%d-%d-%d",&year2,&month2,&day2);

givendate=day2+(100\*month2)+(10000\*year2);

}

Τέλος, καλούμε τον αλγόριθμο δυικής αναζήτησης παρεμβολής, ή τη βελτιωμένη εκδοχή του, και μετράμε τον χρόνο εκτέλεσης.

Για τον αλγόριθμο Binary Interpolation Search εκτελούμε μία συνάρτηση η οποία παίρνει ως όρισμα τον πίνακα στον οποίο θα γίνει η αναζήτηση, το μέγεθος του πίνακα και την ημερομηνία που θα αναζητήσουμε, δηλαδή αυτή που έχει δοθεί από τον χρήστη. Αρχικά, βάζουμε ως άκρα left και right τις τιμές 0 και το μέγεθος του πίνακα μειωμένο κατά 1 αντίστοιχα, υπολογίζουμε το μέγεθος ως right-left και ορίζουμε και το next μας ως . Όσο η ημερομηνία date που μας έχει δοθεί από τον χρήστη είναι διαφορετική από το arr[next].date και η μεταβλητή found, που αρχικοποιείται με 0 και γίνεται 1 μόνο αν βρεθεί η ημερομηνία με απευθείας αναζήτηση για size<=3, είναι διαφορετική από 1 θα εκτελείται ένας βρόγχος επανάληψης while στη διάρκεια του οποίου το size θα αρχικοποιείται με right-left και αν είναι μικρότερο ή ίσο με 3 θα εκτελεστεί το ακόλουθο if στο οποίο γίνεται απευθείας αναζήτηση:

if(size<=3)

{

found=0;

j=left;

while((found!=1) && (j<=right))

{

if(arr[j].date==date)

{

found=1;

}

else

{

j=j+1;

}

}

}

Αλλιώς αν η δοθείσα ημερομηνία είναι μεγαλύτερη ή ίση με την arr[next].date θα εκτελείται το ακόλουθο if στο οποίο θα κινούμαστε με βήματα μέχρι να μην ισχύει date>arr[next+(i\*d)].date, όπου i ο αριθμός των βημάτων που θα χρειαστούν και d είναι ίσο με , και μετά θα δώσουμε νέες τιμές στο δεξί και το αριστερό άκρο αναζήτησης μας:

if(date>=arr[next].date)

{

i=0;

d=(int)sqrt(size);

while((date>arr[next+(i\*d)].date)&&(next+i\*d<num))

{

i=i+1;

}

right=next+i\*d;

left=next+(i-1)\*d;

}

Αλλιώς αν η δοθείσα ημερομηνία είναι μικρότερη της arr[next].date θα εκτελείται το ακόλουθο if στο οποίο θα κινούμαστε με βήματα μέχρι να μην ισχύει date<arr[next-(i\*d)].date, όπου i ο αριθμός των βημάτων που θα χρειαστούν και d είναι ίσο με , και μετά θα δώσουμε νέες τιμές στο δεξί και το αριστερό άκρο αναζήτησης μας:

else if(date<arr[next].date)

{

i=0;

d=(int)sqrt(size);

while((date<arr[next-(i\*d)].date)&&(next-i\*d>0))

{

i=i+1;

}

right=next-(i-1)\*d;

left=next-i\*d;

}

Τέλος, υπολογίζουμε τη νέα τιμή του next σύμφωνα με τα right και left που προέκυψαν από παραπάνω και αν χρειαστεί επαναλαμβάνουμε το while loop. Σε περίπτωση που το found έχει γίνει 1 ή που το arr[next].date είναι ίσο με το date, θα εμφανιστεί στον χρήστη ο όγκος των συναλλαγών για την ημερομηνία που έδωσε και τη θέση της ημερομηνίας στο αρχείο που διαβάσαμε. Για να ελέγξουμε την πιθανότητα η δοθείσα ημερομηνία να μην ανήκει στο αρχείο έχουμε προσθέσει μία επιπλέον μεταβλητή η οποία ονομάζεται wrong. Παρατηρήσαμε εκτελώντας τον κώδικα ότι αν βάλουμε μία ημερομηνία που δεν ανήκει στο αρχείο είναι πιθανό 1.να βγαίνει εξαρχής αρνητικό το next ή 2.να επαναλαμβάνεται σε πολλές επαναλήψεις του εξωτερικού while το ίδιο next ή το next να είναι μικρότερο του 0 ή μεγαλύτερο του μεγέθους του πίνακα, δηλαδή να ξεπερνά τα όρια του πίνακα. Για την περίπτωση 1 ελέγχει το ακόλουθο τμήμα του κώδικα:

if(next<0)

{

wrong=1;

}

Ενώ για την 2 το εξής κομμάτι του κώδικα:

if(buffer[bc]==buffer[bc-1])

{

counter++;

}

if((counter>=10)||(buffer[bc]<0)||(buffer[bc]>num))

{

wrong=1;

}

Όπου buffer ένας μικρός πίνακας που αποθηκεύει τις τιμές του next και counter ένας μετρητής που υπολογίζει πόσες φορές επαναλαμβάνεται ο next. Αν το wrong γίνει ίσο με 1, τότε βγαίνει μήνυμα λάθους που επιβεβαιώνει ότι ο συγκεκριμένος όγκος δεν υπάρχει.

Για τον αλγόριθμο της παραλλαγής του Binary Interpolation Search εκτελούμε μία συνάρτηση που παίρνει ως όρισμα τον πίνακα που θα γίνει η αναζήτηση, το μέγεθος του και την ημερομηνία που δόθηκε από τον χρήστη. Η υλοποίηση του σε γενικές γραμμές είναι ίδια με τον κώδικα που αναλύθηκε παραπάνω για την δυική αναζήτηση παρεμβολής, με μερικές αλλαγές. Αρχικά, σε αυτή την περίπτωση το βήμα όταν η δοθείσα ημερομηνία είναι μεγαλύτερη ή ίση με την arr[next].date δεν θα είναι , αλλά . Επίσης θα αλλάζει μόνο η τιμή του right και έπειτα θα ψάξουμε το m για το οποίο ισχύει (date>=arr[next+((i/2)+m)\*d].date)&&(date<=arr[next+((i/2)+(m+1))\*d].date), όπου d είναι ίσο με , με δυαδική αναζήτηση με άκρα left=next+((i/2)+m)\*d και right=next+((i/2)+(m+1))\*d. Όταν βρούμε το σωστό m, αλλάζουμε τα άκρα right και left, όπως φαίνεται παρακάτω:

if(date>=arr[next].date)

{

i=0;

m=0;

d=(int)sqrt(size);

while(date>arr[next+i\*d].date)

{

if(i==0)

{

i=1;

}

else

{

i=2\*i;

}

}

left=next;

right=next+i\*d;

while(!((date>=arr[next+((i/2)+m)\*d].date)&&(date<=arr[next+((i/2)+(m+1))\*d].date)))

{

left=next+((i/2)+m)\*d;

right=next+((i/2)+(m+1))\*d;

mid=(int)((right-left)/2);

if(date<arr[mid].date)

{

m++;

}

if(date>=arr[mid].date)

{

m--;

}

}

left=next+((i/2)+m)\*d;

right=next+((i/2)+(m+1))\*d;

}

Ομοίως αντιμετωπίζεται και η περίπτωση που η δοθείσα ημερομηνία είναι μικρότερη της arr[next].date, μόνο που σε αυτή την περίπτωση θα κρατήσουμε σταθερή την τιμή του right και θα αλλάξουμε μόνο το left και έπειτα θα ψάξουμε το m για το οποίο ισχύει (date>=arr[next-((i/2)+m+1)\*d].date)&&(date<=arr[next-((i/2)+m)\*d].date), όπου d είναι ίσο με , με δυαδική αναζήτηση με άκρα left=next-((i/2)+m+1)\*d και right=next-((i/2)+m)\*d. Όταν βρούμε το σωστό m, αλλάζουμε τα άκρα right και left, όπως φαίνεται παρακάτω:

else if(date<arr[next].date)

{

i=0;

m=0;

d=(int)sqrt(size);

while(date>arr[next-i\*d].date)

{

if(i==0)

{

i=1;

}

else

{

i=2\*i;

}

}

left=next-i\*d;

right=next;

while(!((date>=arr[next-((i/2)+m+1)\*d].date)&&(date<=arr[next-((i/2)+m)\*d].date)))

{

right=next-((i/2)+m)\*d;

left=next-((i/2)+m+1)\*d;

mid=(right-left)/2;

if(date<arr[mid].date)

{

m++;

}

if(date>=arr[mid].date)

{

m--;

}

}

right=next-((i/2)+m)\*d;

left=next-((i/2)+m+1)\*d;

}

Τέλος, αλλάζουμε το next, όπως στην περιγραφή του προηγούμενου αλγορίθμου, και αν χρειαστεί επαναλαμβάνουμε το while loop. Σε περίπτωση που το found έχει γίνει 1 ή που το arr[next].date είναι ίσο με το date, θα εμφανιστεί στον χρήστη ο όγκος των συναλλαγών για την ημερομηνία που έδωσε και τη θέση της ημερομηνίας στο αρχείο που διαβάσαμε. Αλλιώς αν η ημερομηνία που δόθηκε δεν υπάρχει στο αρχείο, με ελέγχους ίδιους με τον προηγούμενο αλγόριθμο, θα εμφανιστεί κατάλληλο μήνυμα λάθους.

Εκτελώντας τον αλγόριθμο της δυικής αναζήτησης παρεμβολής για το αρχείο agn.us.txt πολλές φορές συγκεντρώσαμε τους ακόλουθους χρόνους: 0.000994sec, 0.001033sec, 0.000434sec, 0.000566sec, 0.000997sec, 0.000998sec, 0.000806sec, 0.001033sec, 0.000627sec και 0.000955sec.

Από αυτές τις μετρήσεις συμπεραίνουμε πως ο μέσος χρόνος είναι ίσος με 0.0008443sec, ενώ ο χειρότερος είναι ίσος με 0.001033sec.

Εκτελώντας τον αλγόριθμο της δυικής αναζήτησης παρεμβολής για το αρχείο ale.us.txt πολλές φορές συγκεντρώσαμε τους ακόλουθους χρόνους: 0.000296sec, 0.000996sec, 0.000559sec, 0.000982sec, 0.001012sec, 0.000512sec, 0.000679sec, 0.000413sec, 0.000300sec και 0.000999sec.

Από αυτές τις μετρήσεις συμπεραίνουμε πως ο μέσος χρόνος είναι ίσος με 0.0006748sec, ενώ ο χειρότερος είναι ίσος με 0.001012sec.

Εκτελώντας τον αλγόριθμο της δυικής αναζήτησης παρεμβολής για το αρχείο ainv.us.txt πολλές φορές συγκεντρώσαμε τους ακόλουθους χρόνους: 0.000999sec, 0.000343sec, 0.000259sec, 0.000997sec, 0.000329sec, 0.000300sec, 0.001247sec, 0.000365sec, 0.000882sec και 0.001069sec.

Από αυτές τις μετρήσεις συμπεραίνουμε πως ο μέσος χρόνος είναι ίσος με 0.000679sec, ενώ ο χειρότερος είναι ίσος με 0.001247sec.

Αντίστοιχα, εκτελώντας την παραλλαγή του για το αρχείο ainv.us.txt πολλές φορές συγκεντρώσαμε τους ακόλουθους χρόνους: 0.000256sec, 0.001022sec, 0.000172sec, 0.001004sec, 0.000995sec, 0.000222sec, 0.000999sec, 0.000162sec, 0.000960sec και 0.000996sec.

Από αυτές συμπεραίνουμε πως ο χειρότερος χρόνος είναι ίσος με 0.001022sec και είναι μικρότερος από τον αντίστοιχο του BIS.

Αντίστοιχα, εκτελώντας την παραλλαγή του για το αρχείο ale.us.txt πολλές φορές συγκεντρώσαμε τους ακόλουθους χρόνους: 0.000530sec, 0.000959sec, 0.000953sec, 0.001005sec, 0.000911sec, 0.001009sec, 0.000998sec, 0.000999sec, 0.001005sec και 0.000997sec.

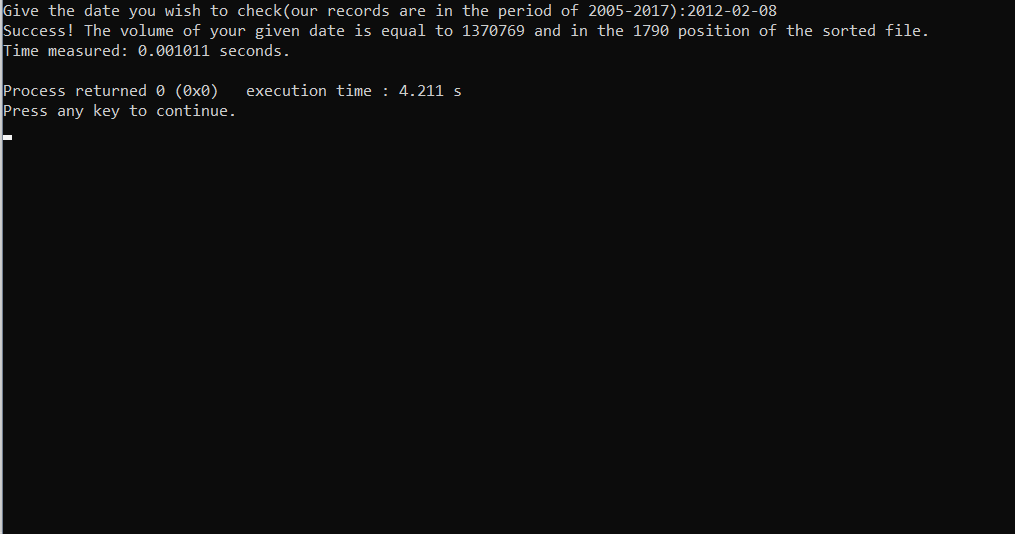
Από αυτές συμπεραίνουμε ότι ο χειρότερος χρόνος είναι ίσος με 0.001009sec και είναι μικρότερος από τον αντίστοιχο του BIS.

Αντίστοιχα, εκτελώντας την παραλλαγή του για το αρχείο agn.us.txt πολλές φορές συγκεντρώσαμε τους ακόλουθους χρόνους: 0.000996sec, 0.000960sec, 0.001006sec, 0.000953sec, 0.001006sec, 0.001002sec, 0.000948sec, 0.001002sec, 0.001003sec και 0.000940sec.

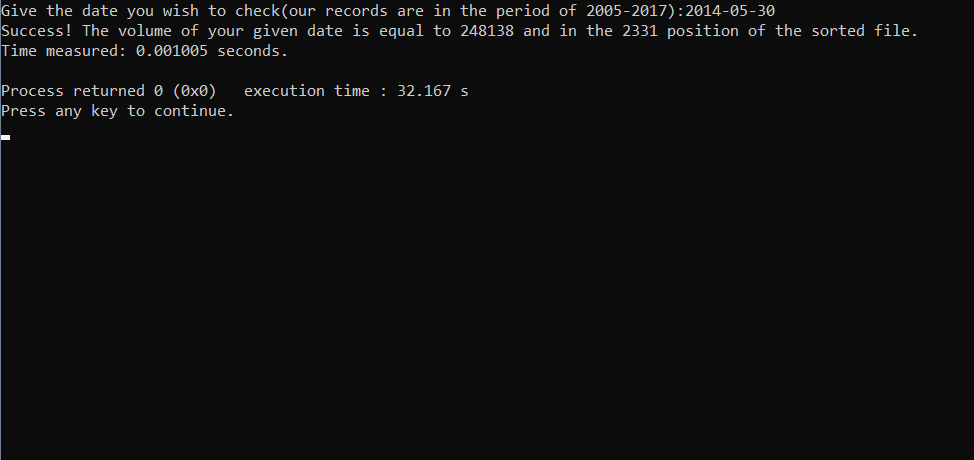
Από αυτές συμπεραίνουμε ότι ο χειρότερος χρόνος είναι ίσος με 0.001006sec και είναι μικρότερος από τον αντίστοιχο του BIS.

Μερικά στιγμιότυπα από την εκτέλεση του BIS είναι τα ακόλουθα:

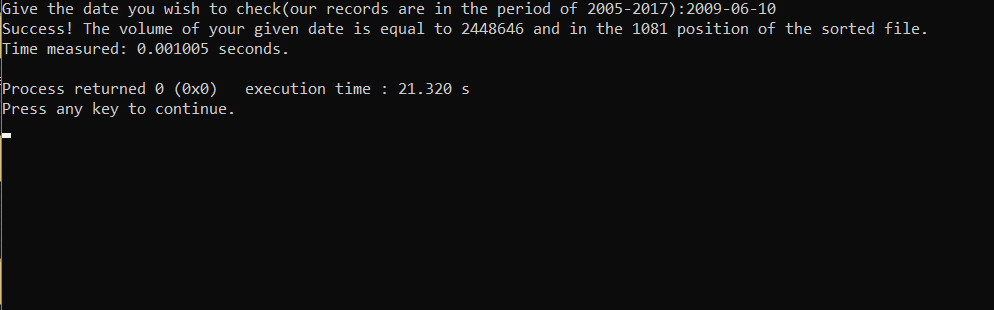
Για το agn.us.txt:



Για το ale.us.txt:

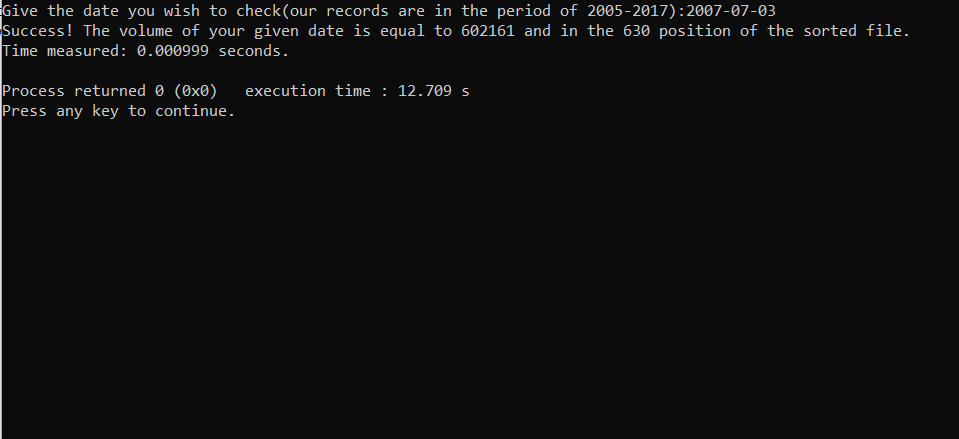


Για το ainv.us.txt:



Μερικά στιγμιότυπα από την εκτέλεση της παραλλαγής του BIS είναι τα ακόλουθα:

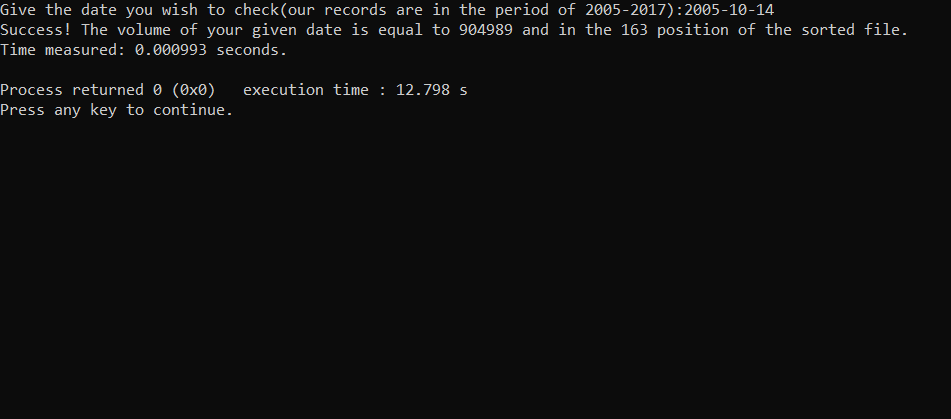
Για το agn.us.txt:



Για το ale.us.txt:



Για το ainv.us.txt:



**Part 2(A):**

Η συγκεκριμένη εφαρμογή υλοποιήθηκε στο Code::Blocks και για όσα στιγμιότυπα ελέγχτηκε ήταν πλήρως λειτουργική. Επίσης, το δέντρο υλοποιήθηκε ως AVL.

Αρχικά, για να δημιουργήσουμε το δέντρο από το αρχείο agn.us.txt δημιουργήσαμε τις ακόλουθες βοηθητικές συναρτήσεις:

1.Τη συνάρτηση height, η οποία επιστρέφει την μεταβλητή height του κόμβου, δηλαδή το ύψος του.

2.Τη συνάρτηση rightRotate, η οποία εκτελεί δεξιά περιστροφή στο δέντρο για έναν κόμβο x.

3.Τη συνάρτηση leftRotate, η οποία εκτελεί αριστερή περιστροφή στο δέντρο για έναν κόμβο x.

4.Τη συνάρτηση Balance, η οποία επιστρέφει την υψοζύγιση του κόμβου x, γιατί κατά τη δημιουργία του δέντρου, ή στη διάρκεια των διαγραφών, πρέπει να την υπολογίζουμε για να σιγουρευτούμε ότι παίρνει τιμές ανάμεσα στο -1 και στο 1.

Έπειτα, δημιουργούμε μία συνάρτηση insert, την οποία θα καλέσουμε στη main με ένα for loop για όλα τα στοιχεία του αρχείου. Σε αυτή, αρχικά δημιουργούμε τον κόμβο όπως φαίνεται παρακάτω:

if (x == NULL)

{

struct node\* x = (struct node\*)malloc(sizeof(struct node));

x->date=dt;

x->volume=v;

x->left=NULL;

x->right=NULL;

x->height=1;

return x;

}

Και μετά ελέγχουμε τη σχέση της ημερομηνίας που προσθέτουμε με την ημερομηνία της ρίζας. Αν είναι μεγαλύτερη καλείται το insert για το δεξί υποδέντρο, ενώ αν είναι μικρότερη για το αριστερό. Ίσες ημερομηνίες δεν επιτρέπονται στα δυαδικά δέντρα αναζήτησης. Αυτά γίνονται με τον ακόλουθο κώδικα:

if (dt < x->date)

{

x->left = insert(x->left,dt,v);

}

else if (dt > x->date)

{

x->right = insert(x->right,dt,v);

}

else

{

return x;

}

Στη συνέχεια υπολογίζουμε το ύψος του κόμβου x και ελέγχουμε την υψοζύγιση με τη συνάρτηση Balance. Αν το balance είναι μεγαλύτερο του 1 και η ημερομηνία μας είναι μεγαλύτερη από την ημερομηνία του δεξιού παιδιού του x, τότε κάνουμε απλή αριστερή περιστροφή, όπως φαίνεται παρακάτω:

if ((balance > 1) && (dt > x->right->date))

{

return leftRotate(x);

}

Αν το balance είναι μικρότερο του -1 και η ημερομηνία μας είναι μικρότερη από την ημερομηνία του αριστερού παιδιού του x, τότε κάνουμε απλή δεξιά περιστροφή, όπως φαίνεται παρακάτω:

if ((balance < -1) && (dt < x->left->date))

{

return rightRotate(x);

}

Αν το balance είναι μεγαλύτερο του 1 και η ημερομηνία μας είναι μικρότερη από την ημερομηνία του δεξιού παιδιού του x, τότε κάνουμε δεξιά περιστροφή στο δεξί παιδί του x και αριστερή περιστροφή στο x, όπως φαίνεται παρακάτω:

if ((balance > 1) && (dt < x->right->date))

{

x->right = rightRotate(x->right);

return leftRotate(x);

}

Τέλος, αν το balance είναι μικρότερο του -1 και η ημερομηνία μας είναι μεγαλύτερη από την ημερομηνία του αριστερού παιδιού του x, τότε κάνουμε αριστερή περιστροφή στο αριστερό παιδί του x και δεξιά περιστροφή στο x, όπως φαίνεται παρακάτω:

if ((balance < -1) && (dt > x->left->date))

{

x->left = leftRotate(x->left);

return rightRotate(x);

}

Επίσης, έχουμε δημιουργήσει μία συνάρτηση inOrder για την εμφάνιση του δέντρου με ενδοδιατεταγμένη διάσχιση, εμφανίζοντας πρώτα τα αριστερά υποδέντρα, μετά τη ρίζα και τέλος τα δεξιά υποδέντρα, όπως φαίνεται παρακάτω:

void inOrder(struct node \*root)

{

if(root != NULL)

{

inOrder(root->left);

int numb[8], number,dig=0;

number=root->date;

while(dig<8)

{

numb[dig] = number % 10;

number=number/10;

dig++;

}

printf("%d%d%d%d-%d%d-%d%d", numb[7],numb[6],numb[5],numb[4],numb[3],numb[2],numb[1],numb[0] );

printf(" (Balance=%d) ",Balance(root));

inOrder(root->right);

}

}

Ακόμη, έχουμε μία συνάρτηση searchInAVL με την οποία βρίσκουμε τον όγκο μίας δοσμένης ημερομηνίας. Αν η ημερομηνία είναι μικρότερη από της ρίζας κάνουμε αναζήτηση στο αριστερό υποδέντρο, ενώ αν είναι μεγαλύτερη στο δεξιά, μέχρι να τη βρούμε, όπως φαίνεται παρακάτω:

void searchInAVL(int date, struct node\* root)

{

if(root!=NULL)

{

if (date<root->date)

{

return searchInAVL(date,root->left);

}

else if (date>root->date)

{

return searchInAVL(date,root->right);

}

else if (date==root->date)

{

printf("The volume of your given date is %d\n",root->volume);

}

}

else

{

printf("Not valid date\n");

}

}

Όμοια, έχουμε μία συνάρτηση που αλλάζει τον όγκο για μία δοσμένη ημερομηνία, βρίσκοντας αυτή την ημερομηνία με παρόμοιο τρόπο όπως παραπάνω και στη συνέχεια αλλάζοντας το πεδίο volume της, όπως φαίνεται παρακάτω:

void changeInAVL(int date, struct node\* root)

{

int volume,pvol;

if(root!=NULL)

{

if (date<root->date)

{

return changeInAVL(date,root->left);

}

else if (date>root->date)

{

return changeInAVL(date,root->right);

}

else if (date==root->date)

{

pvol=root->volume;

printf("Give the volume you want to write in this date: \n");

scanf("%d",&volume);

root->volume=volume;

printf("The previous volume of your given date was %d and now is equal to %d.\n",pvol,root->volume);

}

}

else

{

printf("Not valid date\n");

}

}

Τέλος, έχουμε μία συνάρτηση Delete, η οποία διαγράφει έναν κόμβο από το AVL μας. Για αυτό ελέγχουμε αν η ημερομηνία του κόμβου που θέλουμε να διαγράψουμε είναι μεγαλύτερη ή μικρότερη από αυτή της ρίζας. Αν είναι μικρότερη καλούμε τη συνάρτηση Delete για το αριστερό υποδέντρο, ενώ αν είναι μεγαλύτερη για το δεξί, όπως παρακάτω:

if ( dt < root->date )

{

root->left = Delete(root->left, dt);

}

else if( dt > root->date )

{

root->right = Delete(root->right, dt);

}

Αν είναι ίση τότε ελέγχουμε αν έχει δύο, ένα ή κανένα παιδί. Αν έχει ένα ή κανένα παιδί τότε ελέγχει αν είναι στα δεξιά ή στα αριστερά του και αντιγράφει σε μία μεταβλητή temp τα περιεχόμενα του, ενώ αν δεν έχει θέτει την temp ίση με τον κόμβο προς διαγραφή, όπως φαίνεται παρακάτω:

if( (root->left == NULL) || (root->right == NULL) )

{

struct node \*temp;

if(root->left==NULL)

{

temp=root->left;

}

else if(root->right==NULL)

{

temp=root->right;

}

if (temp == NULL)

{

temp = root;

root = NULL;

}

else

{

\*root = \*temp;

}

free(temp);

}

Αν έχει δύο παιδιά τότε παίρνουμε το μικρότερο στοιχείο, αντιγράφουμε τα περιεχόμενα του στον κόμβο που θέλουμε να διαγράψουμε και μετά διαγράφουμε τον κόμβο που είχε το μικρότερο στοιχείο, όπως φαίνεται παρακάτω:

else

{

struct node\* current = root->right;

while (current->left != NULL)

{

current = current->left;

}

root->date = current->date;

root->right = Delete(root->right, current->date);

}

Έπειτα, αλλάζουμε το ύψος του κόμβου και παίρνουμε το balance του. Αν δεν είναι -1, 0 ή 1 τότε κάνουμε τις κατάλληλες περιστροφές, όπως στην περίπτωση του insert.

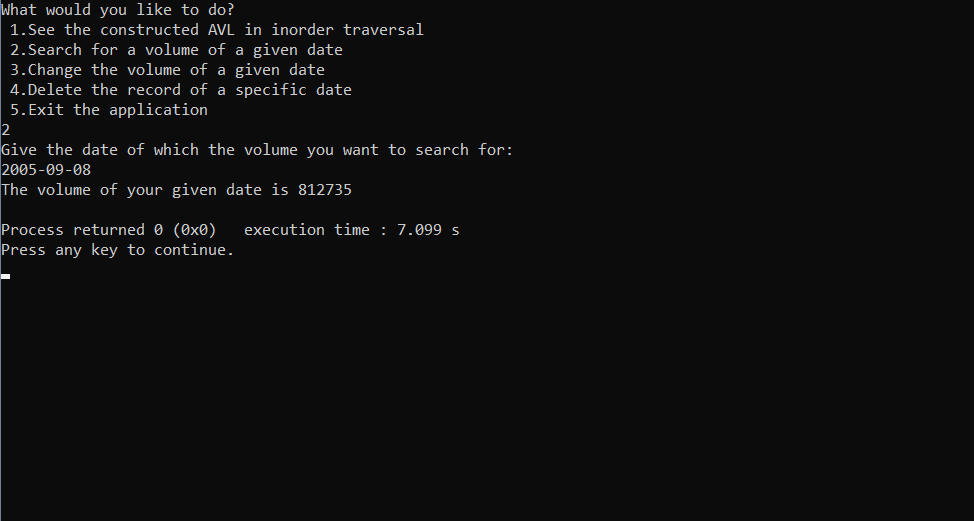
Μερικά στιγμιότυπα από την λειτουργία του κώδικα είναι τα ακόλουθα:

Για την περίπτωση 1, απεικόνιση με ενδοδιατεταγμένη διάσχιση:

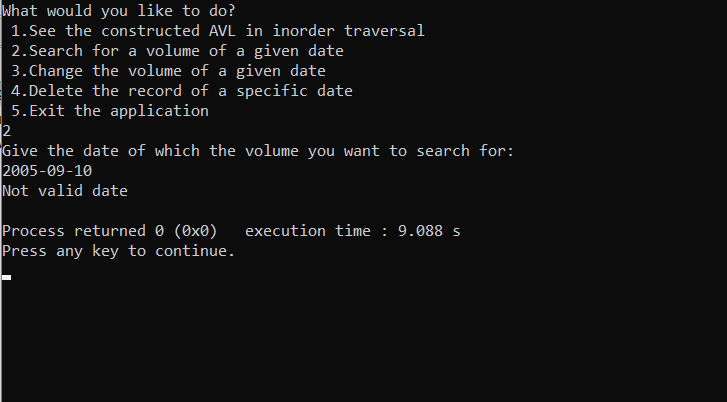


Για την περίπτωση 2, αναζήτηση όγκου με βάση μία δοσμένη ημερομηνία:

Αν υπάρχει η ημερομηνία:

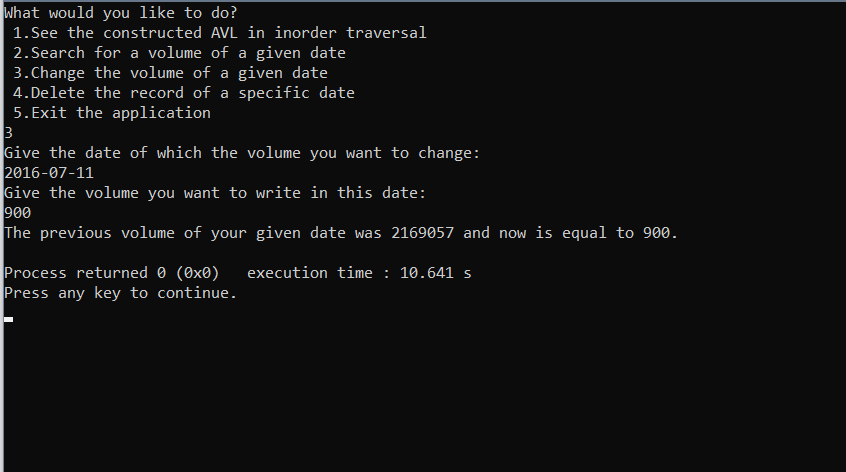


Αν δεν υπάρχει η ημερομηνία:

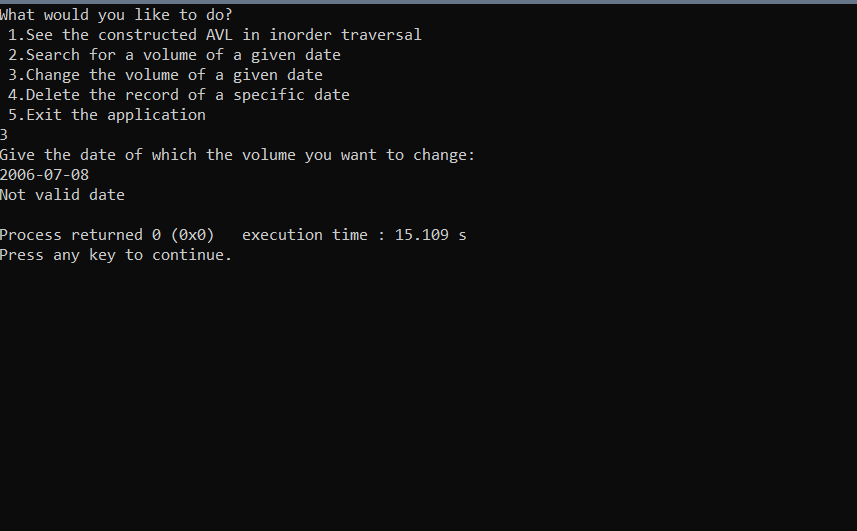


Για την περίπτωση 3, τροποποίηση όγκου συναλλαγών για δοσμένη ημερομηνία:

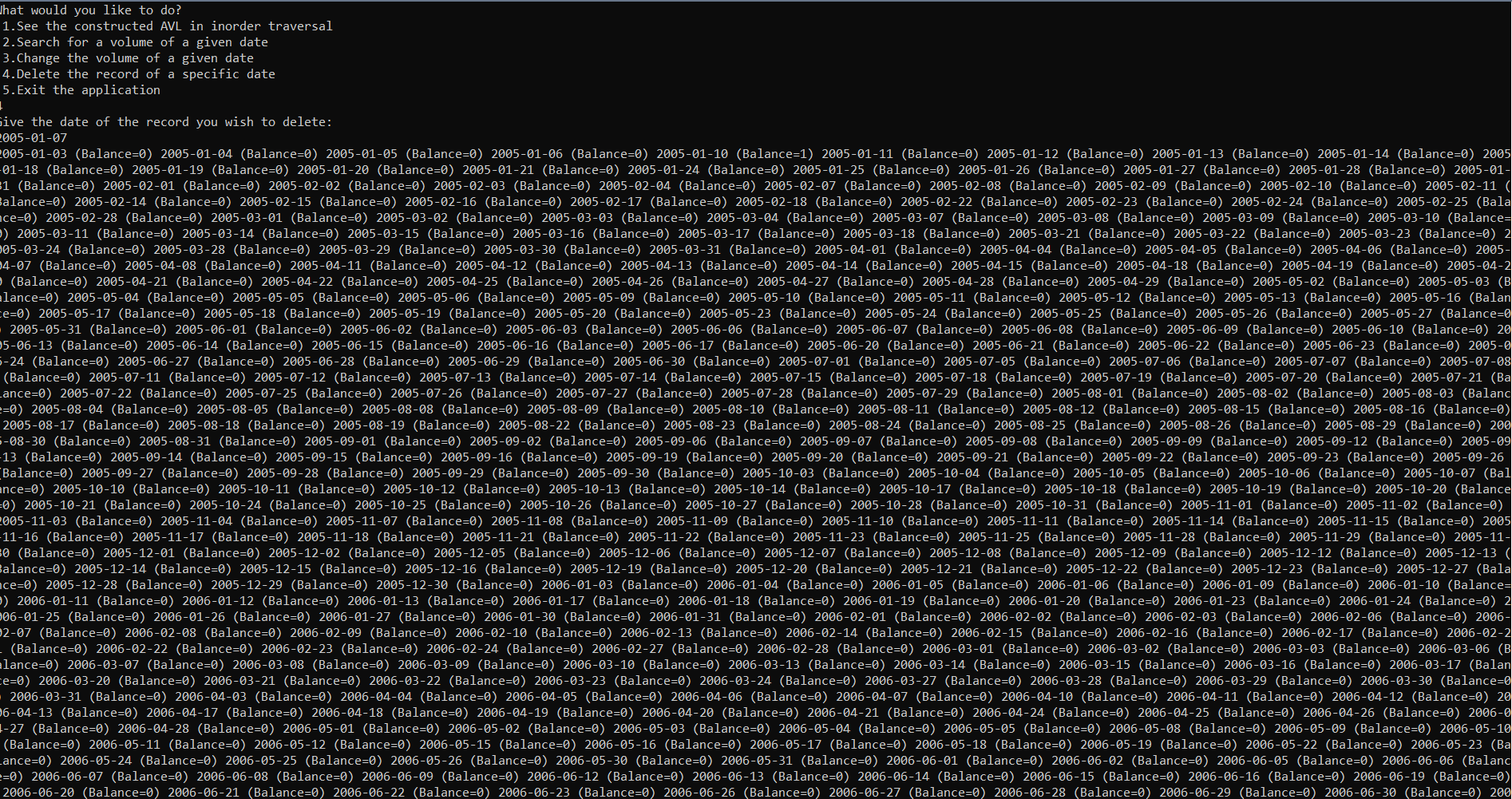
Αν υπάρχει η ημερομηνία:



Αν δεν υπάρχει η ημερομηνία:

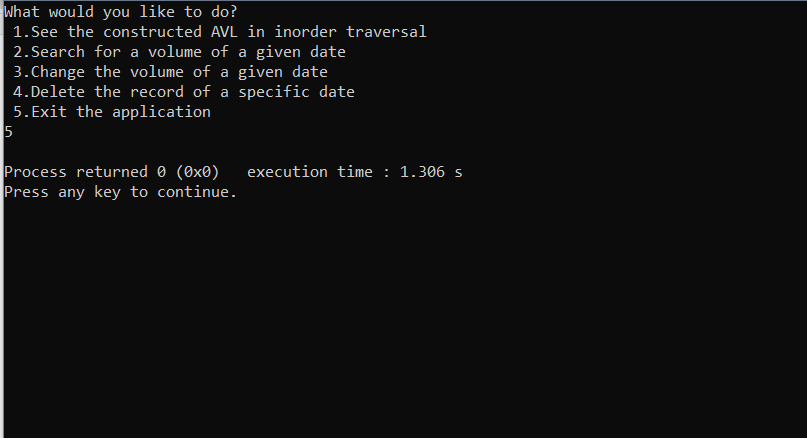


Για την περίπτωση 4, διαγραφή μίας εγγραφής για δοσμένη ημερομηνία:



Μετά από τη διαγραφή καλούμε ενδοδιατεταγμένη διάσχιση για να δείξουμε ότι δεν υπάρχει πλέον αυτή η εγγραφή.

Για την περίπτωση 5, έξοδος από την εφαρμογή:



**Part 2(B):**

Για να διαβάζεται το AVL με βάση τον όγκο συναλλαγών μετοχής, αρχικά προσθέτουμε δύο ακόμη πεδία στη δομή node, το count για να μετράει πόσα στοιχεία του αρχείου έχουν τον ίδιο όγκο με κάποιο κόμβο, αφού δεν επιτρέπεται στο AVL εισαγωγή τιμής ίδιας με κάποια υπάρχουσα, και το days για να αποθηκεύονται εκεί οι ημερομηνίες με τον ίδιο όγκο.

Οι συναρτήσεις height, rightRotate, leftRotate και Balance παραμένουν ίδιες. Στην insert, όμως, όταν δημιουργούμε έναν νέο κόμβο αρχικοποιούμε και το count με 1 και φτιάχνουμε ένα δυναμικό πίνακα στου οποίου την πρώτη θέση αποθηκεύουμε την ημερομηνία που προσθέτουμε, όπως φαίνεται παρακάτω:

if (x == NULL)

{

struct node\* x = (struct node\*)malloc(sizeof(struct node));

x->date=dt;

x->volume=v;

x->left=NULL;

x->right=NULL;

x->height=1;

x->count=1;

x->days = (int\*)malloc(x->count\*sizeof(int));

x->days[1]=dt;

return x;

}

Έπειτα, ελέγχουμε αν υπάρχει ήδη ο κόμβος και αν ισχύει αυτό τότε αυξάνουμε κατά ένα το count και προσθέτουμε την ημερομηνία αυτού του όγκου στον πίνακα των ημερομηνιών, όπως ακολούθως:

if (v == x->volume)

{

int i=1;

x->count=(x->count)+1;

x->days[x->count]=dt;

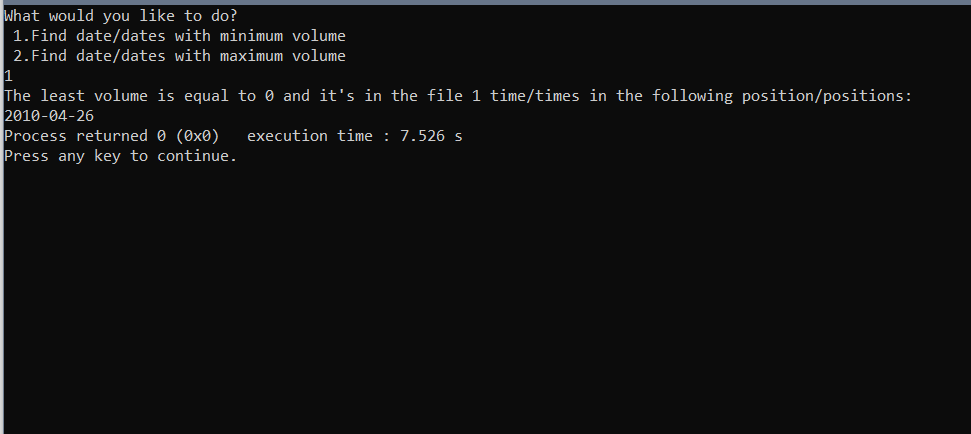
return x;

}

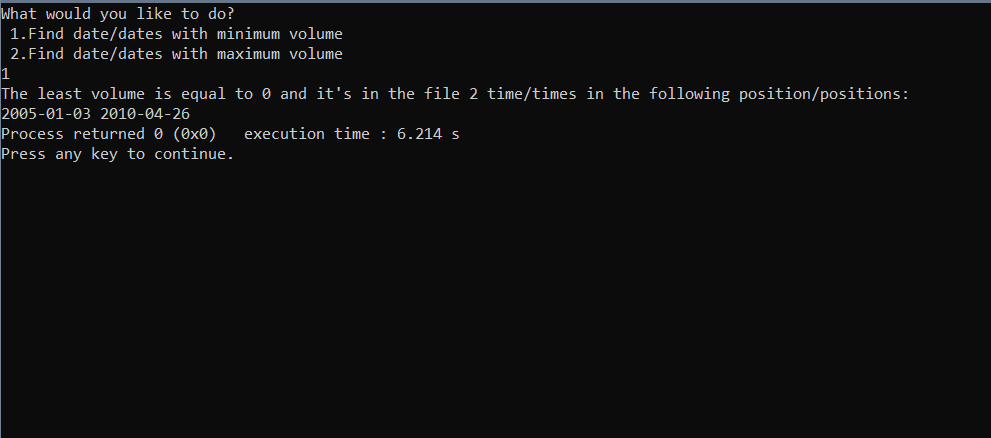
Αν δεν υπάρχει ήδη τον προσθέτουμε όπως στο (Α) υποερώτημα, ελέγχοντας το ύψος, το balance του και κάνοντας τις κατάλληλες περιστροφές.

Τέλος, υπάρχουν δύο επιπλέον συναρτήσεις, η minimumVolume και η maximumVolume, οι οποίες επιτρέπουν τον ελάχιστο και τον μέγιστο όγκο αντίστοιχα, καθώς και ποιες ημερομηνίες έχουν τέτοιο όγκο.

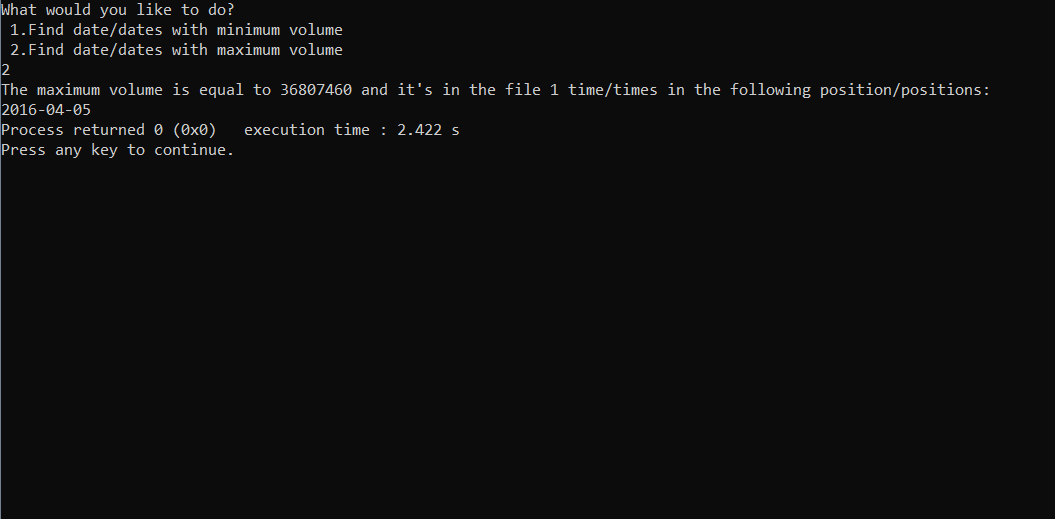
Για το ερώτημα 1, δηλαδή για την εύρεση της ημέρας/των ημερών με την ελάχιστη τιμή όγκου συναλλαγών, το αποτέλεσμα είναι το ακόλουθο:



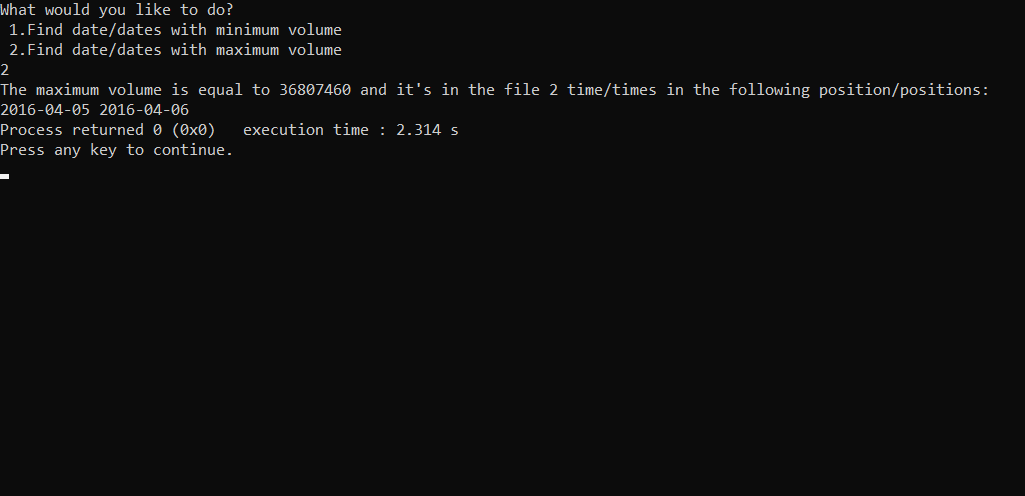
Επειδή ήταν μόνο μία ημερομηνία με ελάχιστο όγκο, αλλάξαμε την τιμή μίας ήδη υπάρχουσας για να δείξουμε ότι λειτουργεί και για περισσότερες ημερομηνίες με κοινό όγκο:



Για το ερώτημα 2, δηλαδή για την εύρεση της ημέρας/των ημερών με την μέγιστη τιμή όγκου συναλλαγών, το αποτέλεσμα είναι το ακόλουθο:



Ομοίως, επειδή ήταν μόνο μία ημερομηνία με μέγιστο όγκο, αλλάξαμε την τιμή μίας ήδη υπάρχουσας για να δείξουμε ότι λειτουργεί και για περισσότερες ημερομηνίες με κοινό όγκο:



**Part 2(C):**

Για τη διαχείριση των δεδομένων χρησιμοποιούμε μία δομή που περιέχει:

* Πίνακα char για τις ημερομηνίες.
* Mία μεταβλητή date για τις ημερομηνίες.
* Μια μεταβλητή για τη διαχείριση των όγκων.
* Και ένα κόμβο next.

Έπειτα, υπάρχει η συνάρτηση :

int ASCIIcalculator(char d[65])

{

int sum=0,i=0;

char date[65];

strcpy(date,d);

while(date[i]!='\0')

{

sum=sum+date[i];

i++;

}

return sum;

}

H oποία θα χρησιμοποιηθεί παρακάτω στον κώδικα για να μπορέσουμε να τροποποιήσουμε την ημερομηνία, όπως ζητάει το ερώτημα. Πιο συγκεκριμένα, η συνάρτηση έχει ως είσοδο μία ημερομηνία, της οποίας κάθε χαρακτήρας (και ο χαρακτήρας «-») θα μετατραπεί σε ASCII value και θα προστεθούν σε έναν συνολικό αριθμό που αποθηκεύεται στη μεταβλητή sum.

Παρακάτω, μπορούμε να δούμε την συνάρτηση void insert(char d[65], int vol), η οποία εισάγει μία τιμή όγκου βάσει την ημερομηνία που της αντιστοιχεί. Βλέπουμε στο παρακάτω απόσπασμα πως η πραγματοποιήση της εισαγωγής περιλαμβάνει και τη hash function, έτσι ώστε το πρόγραμμα να υπολογίσει που θα αποθηκευτεί το στοιχείο :

key=ASCIIcalculator(datee);

i=key%SIZE;

Aμέσως μετά, τροποποιούμε το bucket έτσι ώστε μετα την insertion, το head του προηγούμενου αριθμού να δείχνει στον νέο αριθμό, ενώ το head του νέου αριθμού να δείχνει στον επόμενο:

Node->date=key;

Node->volume=vol;

Node->next=NULL;

if(head[i]==NULL)

{

head[i]=Node;

}

else

{

n=head[i];

while(n->next!=NULL)

{

n=n->next;

}

n->next=Node;

}

Η συνάρτηση void search(char d[65])ψάχνει την δομή για έναν συγκεκριμένο όγκο με είσοδο την αντίστοιχη ημερομηνία. Αν βρει το date επιστρέφει την ημερομηνία και τον όγκο με μήνυμα «Date found and the volume is equal to %d», ενώ αν δεν βρεθεί εμφανίζει το μήνυμα «No such element exists.». Για να βρεθεί η ημερομηνία και έτσι ο όγκος, χρησιμοποιείται και πάλι η hash function, ενώ για τη σύγκριση της ημερομηνίας που δίνει ο χρήστης και την αναζήτηση της αντίστοιχης ημερομηνίας χρησιμοποιούμε το παρακάτω απόσπασμα κώδικα :

key=ASCIIcalculator(dateToCompare);

i=key%SIZE; //hash function

sscanf(dateToCompare,"%d-%d-%d",&year1,&month1,&day1);

dt=day1+100\*month1+10000\*year1;

H συνάρτηση void change(char d[65]) τροποποιεί έναν όγκο (δηλαδή αλλάζει τη τιμή του όγκου)για μια δεδομένη ημερομηνία. Όπως στη προηγούμενη συνάρτηση, δεχόμαστε ως είσοδο μία ημερομηνία και τη μετατρέπουμε σε έναν αριθμό που μπορούμε να διαχειριστούμε, ενώ ταυτόχρονα εκτελούμε τη hash function:

key=ASCIIcalculator(dateToCompare);

i=key%SIZE;

sscanf(dateToCompare,"%d-%d-%d",&year1,&month1,&day1);

dt=day1+100\*month1+10000\*year1;

Αμέσως μετά, ελέγχουμε αν υπάρχει η ημερομηνία (αν δεν υπάρχει εμφανίζεται μήνυμα «No such element exists.», ενώ αν υπάρχει συνεχίζουμε τη συνάρτηση). Στη περίπτωση που η ημερομηνία υπάρχει, εμφανίζουμε το μήνυμα «Date found and the volume is equal to %d», όπου ενημερώνουμε τον χρήστη για τη κατάσταση της ημερομηνίας και του αντίστοιχου όγκου.Έπειτα, ζητάμε από τον χρήστη τη καινούρια τιμή όγκου, την εισάγουμε και εκτυπώνουμε τελικό μήνυμα : «After the changes the new volume is equal to %d».

H συνάρτηση void Delete(char d[65]) είναι αρκετά παρόμοια με την insert(). Πάλι, εκτελούμε τη hash function και λαμβάνουμε την ημερομηνία που δόθηκε από το χρήστη για να τη συγκρίνουμε, ενώ αν δε βρεθεί εμφανίζεται το αντίστοιχο μήνυμα που έχουμε προαναφέρει. Παρακάτω φαίνεται η συνέχεια του κώδικα της συνάρτησης :

{

n=head[i];

sscanf(n->Date,"%d-%d-%d",&year1,&month1,&day1);

givend=day1+100\*month1+10000\*year1;

if((n!=NULL)&&(givend==dt))

{

head[i]=n->next;

free(n);

return;

}

while ((n!=NULL)&&(givend!=dt))

{

prev=n;

n=n->next;

sscanf(n->Date,"%d-%d-%d",&year1,&month1,&day1);

givend=day1+100\*month1+10000\*year1;

}

if(n==NULL)

{

printf("No such element exists.\n");

return;

}

prev->next=n->next;

free(n);

}

Όπως μπορούμε να δούμε, είμαστε στη περίπτωση που η ημερομηνία ήταν σωστή και βρέθηκε. Μετατρέπουμε την ημερομηνία σε διαχειρίσιμη μορφή και διαγράφουμε από το bucket την αντίστοιχη τιμή volume. Έπειτα, λόγω της αφαίρεσης πρέπει να αποκαταστήσουμε την hash chain, άρα το head του προηγούμενου στοιχείου δεν δείχνει πια στο διαγεγραμμένο στοιχείο, αλλά στο επόμενο.Τέλος, εκτελούμε την free() για απελευθέρωση μνήμης.

Στη main, αρχικά ανοίγουμε το αρχείο που ζητείται και χρησιμοποιούμε την strtok() για να πάρουμε τις στήλες που μας ενδιαφέρουν και τις αποθηκεύουμε σε αντίστοιχες δομές. Μάλιστα, την στήλη date την δημιουργούμε μέσω της insert().

Παράλληλα, εκτυπώνουμε το hash table προτού ζητήσουμε κάποια είσοδο από τον χρήστη με το παρακάτω απόσπασμα κώδικα:

printf("\nEntries at index %d: ",jk);

if(head[jk] == NULL)

{

printf("No Hash Entry!");

}

else

{

for(n=head[jk];n!=NULL;n=n->next)

{

printf("%s,%d,volume:%d->",n->Date,n->date,n->volume);

}

printf("\n");

}

Εν συνεχεία, υπάρχει το ζητούμενο μενού με αμυντικο προγραμματισμό για διαφορετική είσοδο από τις προκαθορισμένες.Όπως βλέπουμε παρακάτω οι επιλογές είναι 4 i)Search,ii)Change,iii)Delete,iv)Exit :

printf(" 1.Search volume of a given date\n 2.Change the volume of a given date\n 3.Delete a date\n 4.Exit the application\n");

scanf("%d", &option);

while((option!=1)&&(option!=2)&&(option!=3)&&(option!=4))

{

printf("Not a valid choice, please type a number between 1 to 4\n");

scanf("%d",&option);

}

switch(option)

{

case 1:

printf("Give the date of which the volume you want to search for: \n");

scanf("%s", date);

search(date);

break;

case 2:

printf("Give the date of which the volume you want to change: \n");

scanf("%s", date);

change(date);

break;

case 3:

printf("Give the date of the record you wish to delete: \n");

scanf("%s", date);

Delete(date);

break;

case 4:

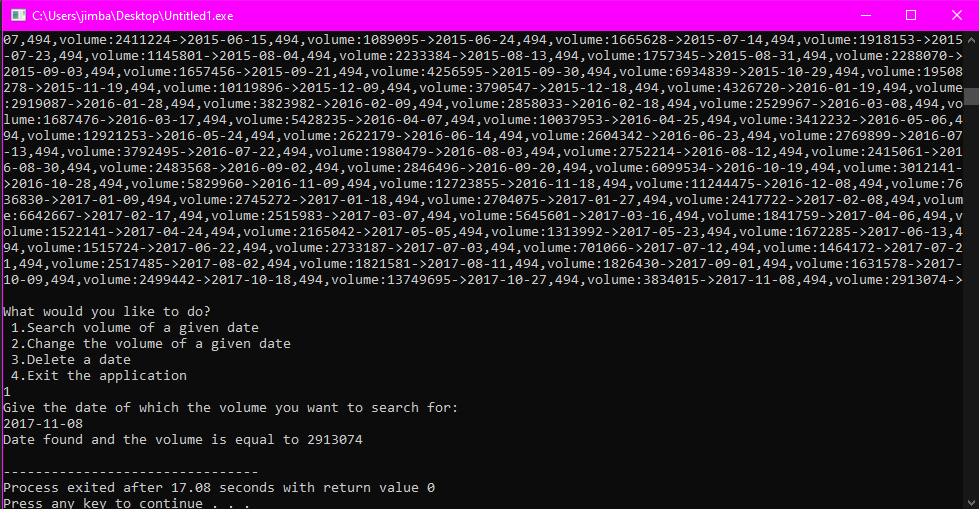
exit(0);

break;

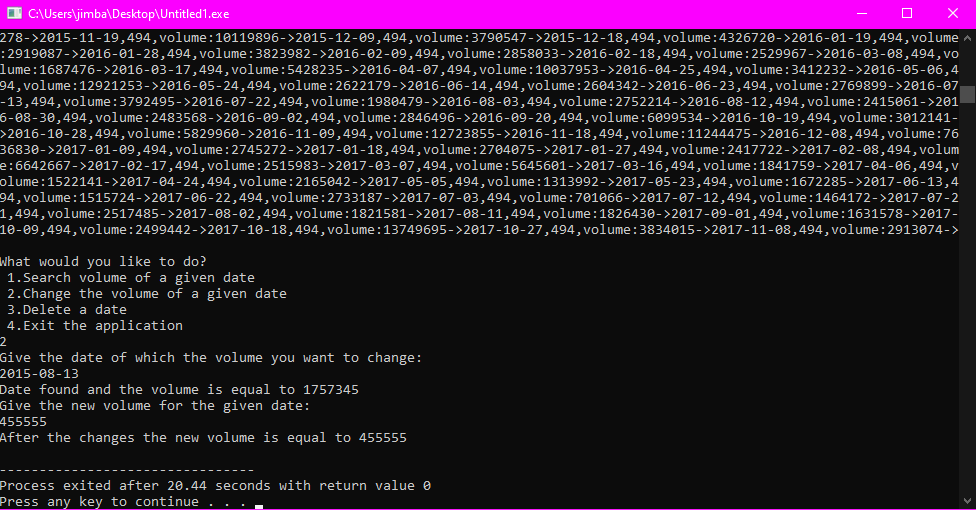
}

Σε καθεμία από τις cases, μετά την επιλογή του case από τον χρήστη, εκτελείται μία print. Ωστόσο στην Delete,αφότου ο χρήστης επιλέξει ποιό date θέλει να διαγράψει, η συνάρτηση εκτυπώνει ξανά το hash table,αλλά αυτή τη φορά η προαναφερθείσα ημερομηνία δεν υπάρχει, το οποίο αναδεικνύει τη σωστή λειτουργία της συνάρτησης.

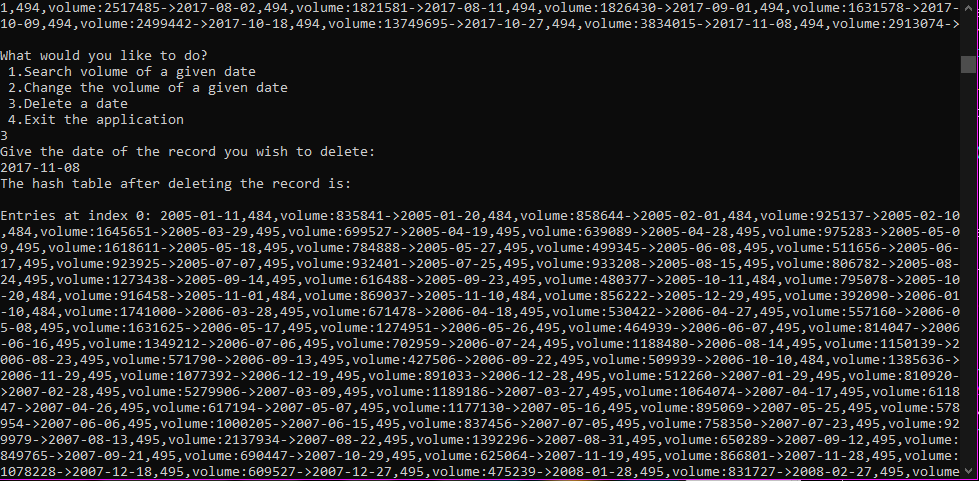
Στιγμιότυπο για την επιλογή 1:



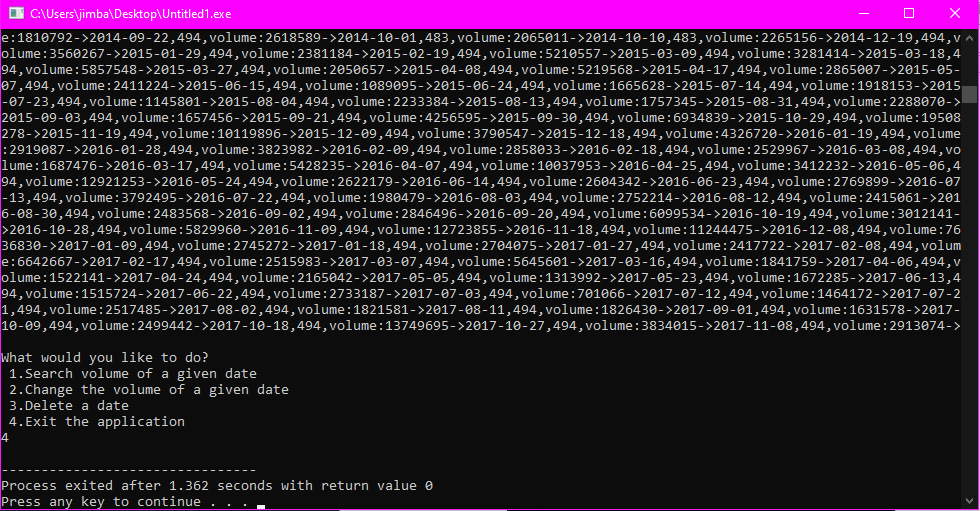
Στιγμιότυπο για την επιλογή 2:



Στιγμιότυπο για την επιλογή 3:



Στιγμιότυπο για την επιλογή 4:



**Part 2(Συνδυασμός των A,B,C):**

Το συγκεκριμένο κομμάτι είναι ο συνδυασμός των 3ων ερωτημάτων του 2ου μέρους, επομένως έχουμε ήδη σχολιάσει του κώδικες. Το εκτελέσιμο αρχείο αποτελείται αρχικά, από όλες τις συναρτήσεις εκτος της main, που χρειαστήκαμε στις 3 ασκήσεις. Συγκεκριμένα, αναφέρονται οι παρακάτω συναρτήσεις:

* int height(struct nodeA \*T)
* int Balance(struct nodeA \*T)
* void inOrder(struct nodeA \*root)
* void searchInAVL(int date, struct nodeA\* root)
* void changeInAVL(int date, struct nodeA\* root)
* int heightB(struct nodeB \*T)
* int BalanceB(struct nodeB \*T)
* void minimumVolumeB(struct nodeB\* x)
* void maximumVolumeB(struct nodeB\* x)
* int ASCIIcalculator(char d[65])
* void insertC(char d[65], int vol)
* void searchC(char d[65])
* void changeC(char d[65])
* void DeleteC(char d[65])

και τα παρακάτω structs:

* struct nodeA
* struct nodeB
* struct nodeC
* struct nodeA \*rightRotate(struct nodeA \*x)
* struct nodeA \*leftRotate(struct nodeA \*x)
* struct nodeA\* insert(struct nodeA\* x, int dt, int v)
* struct nodeA\* Delete(struct nodeA \* root, int dt)
* struct nodeB \*rightRotateB(struct nodeB \*x)
* struct nodeB \*leftRotateB(struct nodeB \*x)
* struct nodeB\* insertB(struct nodeB\* x, int dt, int v)
* struct nodeC\* head[SIZE]={NULL};
* struct nodeC\* n;

Στη main, μετά το άνοιγμα του ζητούμενου αρχείου, εκτυπώνεται μία print η οποία δίνει 2 επιλογές στον χρήστη με κατάλληλο μήνυμα:

int choice1,choice2,option;

printf("What would you like to do?\n");

printf(" 1.Load the file in AVL\n 2.Load the file with hashing with chaining\n");

scanf("%d", &choice1);

1) Να γίνει load το αρχείο μέσω AVL.

2) Να γίνει load το αρχείο με αλυσιδωτό hashing.

Aν ο χρήστης επιλέξει τη πρώτη περίπτωση, θα έχει την επιλιογή να διαλέξει ανάμεσα στα actions του πρώτου ερωτήματος ή του δεύτερου. Τέλος, εάν επιλέξει τη δεύτερη περίπτωση, θα έχει την επιλογή να διαλέξει από τα actions του τρίτου ερωτήματος.