PROJECT: “DATA STRUCTURES 2020”

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Στοιχεία φοιτητών			
Επώνυμο	**Όνομα**	**Α.Μ.**	**Έτος σπουδών**
ΑΒΡΑΜΟΠΟΥΛΟΣ	ΜΙΧΑΗΛ	1067451	2^o
ΚΑΡΑΓΙΑΝΝΗΣ	ΑΛΕΞΑΝΔΡΟΣ	1067452	2^o
ΠΑΝΤΑΖΑΡΑΣ	ΕΥΣΤΡΑΤΙΟΣ	1067490	2^o

Περιεχόμενα

Είδη δεδομένων

Κλάση **DataUnit**:

Αποθηκεύει private μεταβλητές για την ημερομηνία, τη θερμοκρασία και την υγρασία κάθε εγγραφής. Περιλαμβάνει έναν constructor και public μεθόδους για get/set.

Κλάση **AvgDataUnit**:

Αποθηκεύει private μεταβλητές για την ημερομηνία και τη μέση θερμοκρασία της κάθε ημέρας. Περιλαμβάνει έναν constructor και public μεθόδους για get/set.

Διάβασμα Αρχείων

Χρησιμοποιούμε το αρχείο **DataReader**. Οι συναρτήσεις που υλοποιούμε είναι:

readTemp: διαβάζει από το αρχείο την θερμοκρασία και το timestamp, δημιουργεί για κάθε εγγραφή ένα αντικείμενο DataUnit και το προσθέτει στο πίνακα – όρισμα.

Ειδικότερα:

- ανοίγει το αρχείο tempm.txt
- διαβάζει γραμμές του αρχείου
- για κάθε γραμμή, παίρνει τα δεδομένα ανάμεσα στους χαρακτήρες «”» και πετάει τα υπόλοιπα
- κατασκευάζει το αντίστοιχο αντικείμενο DataUnit και το προσθέτει στον άδειο πίνακα όρισμα

readHum: παρόμοια υλοποίηση με την readTemp. Οι διαφορές είναι:

- διαβάζει από το αρχείο hum.txt
- δεν κρατάει τα timestamp
- δεν δημιουργεί νέο DataUnit αλλά προσθέτει δεδομένα σε ένα ήδη υπάρχον

readAvgTemp: παίρνει ως όρισμα τον πίνακα με όλα τα δεδομένα timestamp/ θερμοκρασίας και κατασκευάζει έναν πίνακα με αντικείμενα AvgDataUnit.

Ειδικότερα:

- διατρέχει όλες τις εγγραφές του πίνακα-όρισμα
- αθροίζει τις εγγραφές που έχουν την ίδια μέρα στο timestamp, και διαιρεί με το πλήθος για να βρει μέσο όρο
- δημιουργεί ένα AvgDataUnit αντικείμενο για κάθε διαφορετική μέρα, που περιέχει μόνο την ημερομηνία (χωρίς την ώρα) και τη μέση θερμοκρασία
- προσθέτει αυτά στον πίνακα εξόδου

PART I

(1) Δημιουργούμε αρχείο sorting στο οποίο υλοποιούμε ταξινομήσεις.

Mergesort: Υλοποιείται σε 2 μεθόδους, την κύρια **mergeSortTemp** και μια βοηθητική merge. Στην πρώτη, έχουμε 2 περιπτώσεις:

- Βασική (μέγεθος πίνακα = 1) στην οποία επιστρέφεται ουσιαστικά ο ίδιος ο πίνακας εισόδου

- Αναδρομική, στην οποία ο πίνακας χωρίζεται στη μέση, οι 2 υπο-πίνακες που προκύπτουν ταξινομούνται αναδρομικά και συγχωνεύονται με την βοηθητικη συνάρτηση merge.

Merge: συγχωνεύει 2 ταξινομημένους πίνακες. Διατρέχει ταυτόχρονα και τους 2 συγκρίνοντας ένα ζευγάρι κάθε φορά, και προσθέτει το μικρότερο στοιχείο κάθε φορά στον πίνακα εξόδου.

Quicksort: υλοποιείται στη μέθοδο **quickSortTemp**. Επιλέγει ένα pivot στη μέση του πίνακα εισόδου και βάζει δύο δείκτες, έναν στην αρχή και έναν στο τέλος του. Διατρέχει ταυτόχρονα από δεξιά και αριστερά, ψάχνει για αντιστροφές (δηλ. στοιχεία στο δεξιό υποπίνακα που είναι μικρότερα του pivot και στοιχεία στον αριστερό υποπίνακα που είναι μεγαλύτερα του pivot) και τις εναλλάσσει. Αν δεν υπάρχουν τέτοια, ο πίνακας είναι ταξινομημένος, αλλιώς

αναδρομικά καλείται ο quicksort στους 2 υποπίνακες (left pointer – τέλος) και (αρχή – right pointer).

Σύγκριση: Ο mergesort χρειάστηκε κατά μέσο όρο 0,20s για να ταξινομήσει, ενώ ο quicksort μόλις 0,05. Παρότι ο quicksort κάνει ασυμπτωτικά μεγαλύτερο χρόνο από τον mergesort, έχει πολύ καλύτερο χρόνο μέσης περίπτωσης (για την πλειοψηφία των κατανομών).

(2)Προσθέτουμε τους αλγορίθμους στο ίδιο αρχείο sorting.

Heapsort: Υλοποιείται στην **heapSortHum**, με βάση τη βοηθητική συνάρτηση **heapify**. Η **heapSortHum** αρχικά κατασκευάζει το heap καλώντας heapify για κάθε στοιχείο (εκτός των φύλλων που είναι τετριμμένα). Μετά αφαιρεί κάθε φορά τη ρίζα και την προσθέτει στο ταξινομημένο πίνακα και heapify για να «επισκευάσει» το heap.

Heapify: συγκρίνει την ρίζα όρισμα με τα παιδία της, βρίσκει το μέγιστο από τα τρία και αν αυτό δεν είναι η ρίζα, εναλλάσσει τα δύο. Αν χρειαστεί

αλλαγή, σημαίνει ότι ίσως χάλασε η δομή του υποδέντρου και γι’ αυτό καλείται αναδρομικά η heapify σε αυτό.

Countingsort: υλοποιείται στην **countingSortHum**. Ο αλγόριθμος κατασκευάζει έναν πίνακα count που μετρά πόσα από κάθε στοιχείο περιέχει

ο πίνακας όρισμα. Για να μετρήσει, διατρέχει τον πίνακα και αυξάνει σε κάθε στοιχείο την

αντίστοιχη θέση του count. Μετά αθροίζει κάθε εγγραφή του count με τη προηγούμενη θέση ώστε το νούμερο να αντιστοιχεί σε πραγματική θέση του πίνακα εξόδου. Κάθε φορά που γράφεται μια τιμή στην έξοδο, μειώνεται το count κατά 1 για να δείξει σε κενή θέση (αν το στοιχείο εμφανίζεται πολλαπλές φορές).

Σύγκριση: Ο heapsort χρειάστηκε κατά μέσο όρο 0,14s για να ταξινομήσει, ενώ ο counting sort μόλις 0,001s. Φαίνεται καθαρά ο γραμμικός χρόνος χειρότερης περίπτωσης του αλγορίθμου counting sort ενώ ο heapsort διατηρεί παρόμοιο χρόνο με αυτούς του προηγούμενου ερωτήματος.

(3) Οι αλγόριθμοι εύρεσης γράφονται σε νέο αρχείο Searching. Επίσης για τη σωστή λειτουργία τους θα πρέπει να έχει προηγηθεί sort (όπου χρειάζεται) στον πίνακα εισόδου (δεν περιλαμβάνεται στο sort αλλά υπάρχει υλοποιημένη κατάλληλη συνάρτηση).

Binary Search: Υλοποιείται στην binarySearch. Επιλέγει το μεσαίο στοιχείο του πίνακα και το συγκρίνει με το όρισμα. Αν δεν είναι ίσα, αναδρομικά ψάχνει στον κατάλληλο υποπίνακα (αριστερά αν όρισμα<μέσου, δεξιά αν όρισμα>μέσου).

Interpolation Search: Υλοποιείται στην interpolationSearch. Έχει παρόμοια λογική με το binary search όμως αντί να χωρίζει τον πίνακα στη μέση, τον χωρίζει «έξυπνα» σε σημείο που υπολογίζεται με παρεμβολή. Ο υπολογισμός του σημείου απαιτεί σύγκριση timestamp που υλοποιείται σε βοηθητική συνάρτηση compareTimestamps.

compareTimestamps: Η σύγκριση γίνεται με τη βοήθεια βιβλιοθήκης <time.h> και ακολουθεί τα βήματα:

- δημιουργεί άδειο struct tm
- δημιουργεί stringstream από το timestamp όρισμα
- κάνει parse τα δεδομένα του tm με σωστό format
- δημιουργεί αντικείμενο time_t από το tm
- κάνει την ίδια διαδικασία για το δεύτερο timestamp
- συγκρίνει τα δύο time_t και επιστρέφει τη διαφορά τους σε δευτερόλεπτα

Σύγκριση: Ο binary search χρειάστηκε κατά μέσο όρο 0,18ms για να βρει ένα μέσο στοιχείο, ενώ ο interpolation search 1,20ms. Οι δύο αλγόριθμοι κάνουν παρόμοιο χρόνο παρότι ο binary είναι O(logn) και ο interpolation O(n). Με δεδομένα σε ομοιόμορφη κατανομή ο interpolation γίνεται O(loglogn) όμως στα δικά μας δεδομένα αυτό δεν συμβαίνει επαρκώς. Αυτή η απόκλιση από την κατανομή, και (ίσως) μια μεγαλύτερη σταθερά

πολλαπλασιασμού της πολυπλοκότητας καθιστούν τον interpolation αργότερο.

(4) Οι παρακάτω αλγόριθμοι προστίθενται στο αρχείο searching

Binary Interpolation Search: Υλοποιείται στη μέθοδο BIS. Όπως στο interpolation search επιλέγεται με παρεμβολή το σημείο ψαξίματος. Όμως αυτή τη φορά ο δείκτης αυξάνεται κατά ρίζα(μέγεθος πίνακα) κάθε φορά. Η διαδικασία επαναλαμβάνεται μέχρι το μέγεθος να γίνει αρκετά μικρό ώστε το γραμμικό ψάξιμο να απαιτεί λιγότερο χρόνο. Σε κάθε βήμα ελέγχεται ο δείκτης να μην υπερβεί τα όρια του πίνακα και ανανεώνονται τα left, right. Πάλι χρησιμοποιείται η βοηθητική συνάρτηση compareTimestamps.

Βελτίωση: Αλλάζουμε τον τρόπο που αυξάνεται η μεταβλητή-δείκτης. Αντί να αυξάνουμε με πολλαπλάσια της ρίζας(μεγέθους), αυξάνουμε με δυνάμεις του 2 επί ρίζα(μεγέθους). Έτσι επιτυγχάνουμε ασυμπτωτικά λογαριθμικό χρόνο

χειρότερης περίπτωσης, σε αντίθεση με τον τετραγωνικής ρίζας χρόνο της πρώτης εκδοχής.

Σύγκριση: Ο BIS χρειάστηκε κατά μέσο όρο 1,15ms για να βρει ένα μέσο στοιχείο, ενώ η βελτιωμένη έκδοση 1,13ms. Οι δύο χρόνοι είναι σχεδόν οι ίδιοι, πράγμα λογικό αφού οι αλγόριθμοι έχουν ίδια μέση πολυπλοκότητα και πολύ παρόμοια υλοποίηση. Όμως ασυμπτωτικά, για τις χειρότερες κατανομές, η δεύτερη εκδοχή είναι γρηγορότερη.

PART II

(A) Φτιάχνουμε νέα κλάση **BSTNode** που αντιπροσωπεύει έναν κόμβο του δέντρου. Τα αντικείμενά της έχουν μόνο 3 μεταβλητές: αριστερό παιδί, δεξί παιδί και data.

Δημιουργούμε νέο αρχείο **BinarySearchTree** και εκεί υλοποιούμε τις μεθόδους:

sortedArrayToBST: Μετατρέπει ένα ταξινομημένο πίνακα από **AvgDataUnit** σε μορφή BST. Κάθε

φορά επιλέγει το μεσαίο στοιχείο, κατασκευάζει ένα νέο BSTNode με αυτό και το θέτει ως «ρίζα».

Αναδρομικά καλείται στα παιδιά. Το «μεσαίο» κάθε φορά κανονικοποιείται ώστε αν είναι πολλά ίδια στοιχεία στη σειρά να επιλέγεται το τελευταίο. Πρακτικά αυτό σημαίνει ότι στοιχεία ίσα με τη ρίζα καταλήγουν στο αριστερό υποδέντρο.

InOrderTraversal: Ένδο-διατεταγμένη διαπέραση του δέντρου. Εκτελείται σε 3 βήματα:

Αναδρομική διαπέραση του αριστερού υποδέντρου,

διαπέραση της ρίζας,

αναδρομική διαπέραση του δεξιού υποδέντρου.

searchBST: Βρίσκει την εγγραφή του δέντρου με timestamp που δίνεται ως όρισμα. Συγκρίνει κάθε φορά το timestamp της ρίζας με το όρισμα. Αν είναι ίσα, τότε έχει βρεθεί το στοιχείο. Αλλιώς, η σύγκριση μας δείχνει σε πιο υποδέντρο πρέπει να ψάξουμε αναδρομικά. Η συνάρτηση σύγκρισης

timestamp δέχεται μόνο ολόκληρα timestamps οπότε χρειάζεται να προσθέσουμε ώρες/ λεπτά/ δευτερόλεπτα κάθε φορά που θέλουμε να συγκρίνουμε.

BSTdeleteNode: Δέχεται ως είσοδο ένα BSTNode και το διαγράφει από το δέντρο. Για να διαγραφεί ένας κόμβος, πρέπει να βρεθεί ο pointer του parent που το περιέχει και να διαγραφεί.

Γι’ αυτό, η υλοποίηση μοιάζει με αυτήν της search αλλά ελέγχει ένα βήμα παρακάτω. Συγκρίνει κάθε φορά το timestamp της ρίζας με αυτό ορίσματος. Αν είναι όρισμα<ρίζα ελέγχεται το left pointer της ρίζας. Αν είναι όρισμα>ρίζα ελέγχεται το right pointer της ρίζας. Αν είναι ίσα, είμαστε σε οριακή περίπτωση και ο αλγόριθμος τερματίζει.

BSTfindMin: βρίσκει το έλαχιστο στοιχείο του δέντρου – ορίσματος. Αυτό είναι τετριμμένο λόγω της δομής του δέντρου. Κινείται συνέχεια προς τα αριστερά μέχρι το μέγιστο βάθος του δέντρου.

Φτιάχνουμε νέο αρχείο Menus στο οποίο υλοποιούμε την BSTMenu1. Σε αυτήν, συνδυάζουμε τις παραπάνω μεθόδους και την είσοδο του χρήστη, για να φτιάξουμε τις 5 επιλογές:

1. Απλή κλήση της inOrderTraversal

2. Κλήση της searchBST και εκτύπωση του δεδομένου του κόμβου.

3. Κλήση της searchBST και αλλαγή των μεταβλητών του αντικειμένου-αποτελέσματος.

4. Κλήση της searchBST, κλήση της BSTdeleteNode στο αποτέλεσμα. Αν ο κόμβος είχε παιδί κάνουμε εναλλαγή των δύο. Αν είχε δύο παιδιά κάνουμε το ίδιο με το ελάχιστο του δεξιού υποδέντρου (συνάρτηση BSTfindMin).

5. τερματισμός

Στις (2), (3) και (4) ζητάμε από τον χρήστη να εισάγει το timestamp – όρισμα για την εντολή που έχει επιλέξει.

Για μη αποδεκτές τιμές εισόδου ο χρήστης ενημερώνεται με κατάλληλο μήνυμα (αμυντικός προγραμματισμός).

(B) Προσθέτουμε τις παρακάτω μεθόδους στο αρχείο BinarySearchTree:

BSTfindMax: βρίσκει το μέγιστο στοιχείο του δέντρου – ορίσματος. Αυτό είναι τετριμμένο λόγω της δομής του δέντρου. Κινείται συνέχεια προς τα δεξιά μέχρι το μέγιστο βάθος του δέντρου.

BSTShowMin: καλεί την BSTfindMin για να βρει το ελάχιστο. Μετά διατρέχει το δέντρο για να βρει άλλα ελάχιστα και τα εκτυπώνει.

BSTShowMax: καλεί την BSTfindMax για να βρει το μέγιστο. Μετά διατρέχει το δέντρο για να βρει άλλα μέγιστα και τα εκτυπώνει.

Επίσης υλοποιούμε μια quickSortAvg η οποία είναι σχεδόν ίδια με την quickSortTemp αλλά για AvgDataUnit.

Στο αρχείο **Menus** υλοποιούμε την **BSTMenu2**. Αυτήν την φορά προτού καλέσουμε sortedArrayToBST, πρέπει να ταξινομήσουμε τον πίνακα.

Συνδυάζουμε τις παραπάνω μεθόδους και την είσοδο του χρήστη, για να φτιάξουμε τις 3 επιλογές:

1. Απλή κλήση της BSTShowMin
2. Απλή κλήση της BSTShowMax
3. τερματισμός

(Γ) Δημιουργούμε νέο αρχείο **HashTable** που περιέχει όλες τις σχετικές functions.

hashFunction: Η βασική συνάρτηση κατακερματισμού. Αθροίζει όλους τους χαρακτήρες ASCII του timestamp μιας εγγραφής και μετά κάνει modulo με το μέγεθος του hashTable. Μετά από δοκιμές σχετικά με την κατανομή των δεδομένων, καταλήξαμε σε μέγεθος hashTable 11.

createHashTable: Δημιουργεί το hashTable από έναν πίνακα – όρισμα. Ο πίνακας είναι τριπλός δείκτης AvgDataUnit δηλαδή table> bucket> object. Αρχικοποιεί όλες τις τιμές σε NULL και μετά προσθέτει ένα-ένα όλα τα στοιχεία του

πίνακα, σε θέση που ορίζεται από το hash κάθε στοιχείου.

hashSearch: δέχεται είσοδο το αντίστοιχο bucket του πίνακα. Εκεί, εκτελεί γραμμική αναζήτηση για το στοιχείο.

hashDelete: δέχεται είσοδο το αντίστοιχο bucket του πίνακα. Εκεί, εντοπίζει γραμμικά το στοιχείο προς διαγραφή, και το τελευταίο στοιχείο του πίνακα. Αν τα βρεί, εναλλάσσει τα δυο και διαγράφει το στοιχείο.

Στο αρχείο Menus προσθέτουμε την HashMenu. Αυτή συνδυάζει τις παραπάνω μεθόδους και την είσοδο του χρήστη, για να φτιάξει τις 4 επιλογές:

1. Κλήση της hashFunction για να βρούμε το bucket, κλήση της hashSearch για εύρεση μέσα στο bucket και εκτύπωση του δεδομένου

2. Κλήση της hashFunction για να βρούμε το bucket, κλήση της hashSearch για εύρεση μέσα στο bucket και αλλαγή των μεταβλητών του αντικειμένου που βρέθηκε

3. Κλήση της hashFunction για να βρούμε το bucket, κλήση της hashDelete για αφαίρεση του στοιχείου από το bucket

4. τερματισμός

Η είσοδος του χρήστη, όπου απαιτείται, υλοποιείται με αμυντικό προγραμματισμό.

ΕΝΟΠΟΙΗΣΗ

Προκειμένου να ενώσουμε τα παραπάνω, προσθέτουμε δύο νέες συναρτήσεις στο Menus:

BSTMenu: ενώνει τις BSTMenu1, BSTMenu2 και παρέχει στον χρήστη επιλογή ανάμεσα τους.

MainMenu: ενώνει τις BSTMenu, HashMenu και παρέχει επιλογή ανάμεσα σε αυτές.

Κάθε exit ενός μενού οδηγεί στο προηγούμενο.