Εργασία Δομών Δεδομένων 2025

Χρήστος Παπαδόπουλος (Αεμ 4804)
Αντώνιος Σταμάτης (Αεμ 4801)

Γενικά:
Στόχος αυτή της εργασίας ήταν η δημιουργία μιας βιβλιοθήκης δήλωσης, υλοποίησης, επεξεργασίας και εκτύπωσης, 5 κλασικών δομών δεδομένων (σωρός ελαχίστων, σωρός μεγίστων, δυαδικό δένδρο αναζήτησης AVL, γράφημα χωρίς κατευθύνσεις αλλά με βάρη στις ακμές, πίνακας κατακερματισμού). Μέσα από αυτή την εργασία αναπτύξαμε τις προγραμματιστικές μας ικανότητες στην γλώσσα C++, μελετήσαμε και υλοποιήσαμε μια πληθώρα αλγορίθμων και τέλος αντιληφθήκαμε την σημασία χρονομέτρησης και αξιολόγησης της πολυπλοκότητας κάθε αλγορίθμου για την κατανόηση της χρησιμότητάς του σε αντίστοιχα προγράμματα

Για την υλοποίηση του κώδικα της εργασίας χρησιμοποιήσαμε το online GDB το οποίο μας παρείχε διευκόλυνσή στην cloud από κοινού συγγραφή του κώδικα. Για την αναφορά αυτής της εργασία θα την χωρίσουμε στα 3 στάδια υλοποίησης του:

Υλοποίηση
Χωρίζουμε την υλοποίηση της εργασίας στα 3 στάδια ενός αλγορίθμου

1. **Input**

Η είσοδος της πληροφορίας που εισάγεται στο πρόγραμμα περιλαμβάνει

α)Το αρχείο Commands.txt για τον καθορισμό των εκτελέσιμων εντολών της κάθε δομής

β) Τα αρχεία που αντλεί πληροφορίες η commands (π.χ. a.txt, b.txt, κτλ.)

Εκτελώντας την main και εφόσον υπάρχει το αρχείο commands και μπορεί να δημιουργηθεί ένα αρχείο output, η main διαβάζει από το αρχείο γραμμή προς γραμμή και για κάθε γραμμή εκτελεί την CommandAction

γ) Η CommandAction αποτελεί την συνάρτηση που παίρνει ως είσοδο ένα string που αφορρά την εκτέλεση μιας εντολής πάνω σε μία δομή δεδομένων, την αναγνωρίζει και την εκτελεί κατάλληλη εντολή της αντίστοιχης δομής και εκτυπώνοντας στο τέλος το αποτέλεσμα στο output.txt (όποτε αυτό είναι απαραίτητο) και τον ανάλογο χρόνο εκτέλεσης

Χρησιμοποιούμε τις μεταβλητές "first" και "second" για να αποθηκεύσουμε την πρώτη και την δεύτερη λέξη από το commands.txt με slicing, χωρίζοντας δηλαδή την ενιαία γραμμή σε λέξεις με την βοήθεια της stringstream η οποία μας δίνει την δυνατότητα να υλοποιούμε ένα string ως ροή εισόδου

Αυτό βολεύει γιατί αυτές οι δύο λέξεις είναι που μας δίνουν την κατάλληλη πληροφορία για το ποια εντολή και για ποια δομή θα εκτελεστεί.

Η πρώτη λέξη καθορίζει το είδος της εντολής (πχ BUILD, GETSIZE) και η δεύτερη το σε ποια δομή αναφέρετε η εντολή (πχ MINHEAP,AVL) καλώντας έτσι κατάλληλα την κάθε μέθοδο για την αντίστοιχη δομή με την χρήση πολλαπλών if-else

2. **Εκτέλεση**

Όσο αναφορά την εκτέλεση, εύκολα γίνεται αντιληπτό ότι η λειτουργία της κάθε δομής θα πρέπει να δομείτε ξεχωριστά (ανάλογα με την δομή)

2i) Σωρός ελαχίστων- Χρησμεύει για την γρήγορη αναζήτηση του ελάχιστου στοιχείου/ υλοποιεί την σειρά προτεραιότητας κλπ.

Τα στοιχεία της σορού αποθηκεύονται σε μια δυναμική λίστα και έπειτα υλοποιούν τις βασικές λειτουργίες μια σωρού ελαχίστων με βάση τις παρακάτω συναρτήσεις:

Build Min Heap: Διαβάζει διαδοχικά αριθμούς από αρχείο. Κάθε αριθμός που διαβάζει μπαίνει στο τέλος της λίστα αποθήκευσης των κόμβων. Σταδιακά ανεβαίνει προς τα πάνω ώστε να ισχύει η ιδιότητα της Min Heap (parent<=child) και αυξάνει το μέγεθος (size) κατά 1

Get Size Min Heap: Επιστρέφει το μέγεθος της σωρού με βάση τον αριθμό των στοιχείων στην αντίστοιχη λίστα

Find Min Min Heap: Επιστρέφει το ελάχιστο της σωρού που είναι πάντοτε στην ρίζα της σωρού.

Insert Min Heap: Εισαγωγή ενός στοιχείου στην σωρό. Αρχικά εισάγετε στην τελική θέση της λίστας και σταδιακά ανεβαίνει προς τα πάνω όσο είναι μικρότερος από τους προηγούμενους κόμβους της λίστας

Delete Min Heap: Διαγραφή του ελαχίστου (ρίζα) της σωρού. Αντιμεταθέτετέ η ρίζα με το τελευταίο στοιχείο της σωρού, απωθείτε από την σωρό (η παλιά ρίζα) και το στοιχείο που βρίσκεται τώρα στην ρίζα με διαδοχικές αντιμεταθέσεις κατεβαίνει προς τα κάτω όσο ισχύει ότι είναι > του επόμενου κόμβου για να συνεχίσει να ισχύει η ιδιότητα της σωρού ελαχίστων

2ii) Σωρός μεγίστων- Χρησιμεύει στην εύρεση του μέγιστου στοιχείου σε χρόνο O(1)/ την υλοποίηση της HeapSort

Τα στοιχεία της σορού αποθηκεύονται σε μια δυναμική λίστα και έπειτα υλοποιούν τις βασικές λειτουργίες μια σωρού ελαχίστων με βάση τις παρακάτω συναρτήσεις:

Build Max Heap: Διαβάζει διαδοχικά αριθμούς από αρχείο. Κάθε αριθμός που διαβάζει μπαίνει στο τέλος της λίστα αποθήκευσης των κόμβων. Σταδιακά ανεβαίνει προς τα πάνω ώστε να ισχύει η ιδιότητα της Max Heap (parent<=child) και αυξάνει το μέγεθος (size) κατά 1

Get Size Max Heap: Επιστρέφει το μέγεθος της σωρού με βάση τον αριθμό των στοιχείων στην αντίστοιχη λίστα

Find Max Max Heap: Επιστρέφει το μέγιστο στοιχείο της σωρού που είναι πάντοτε στην ρίζα της σωρού.

Insert Max Heap: Εισαγωγή ενός στοιχείου στην σωρό. Αρχικά εισάγετε στην τελική θέση της λίστας και σταδιακά ανεβαίνει προς τα πάνω όσο είναι μεγαλύτερος από τους προηγούμενους κόμβους της λίστας

Delete Max Heap: Διαγραφή του ελαχίστου (ρίζα) της σωρού. Αντιμεταθέτετέ η ρίζα με το τελευταίο στοιχείο της σωρού, απωθείτε από την σωρό (η παλιά ρίζα) και το στοιχείο που βρίσκεται τώρα στην ρίζα με διαδοχικές αντιμεταθέσεις κατεβαίνει προς τα κάτω όσο ισχύει ότι είναι < του επόμενου κόμβου για να συνεχίσει να ισχύει η ιδιότητα της σωρού μεγίστων.

2iii) AVL- Χρησιμοποιείτε για αποδοτική και γρήγορη αναζήτηση στοιχείου σε O(logn)

Δημιουργούμε ένα struct που αφορά τον κάθε κόμβο του δέντρου που περιέχει το στοιχείο του, το ύψος του και τους δείκτες για το αριστερό και το δεξί υποδέντρο. Για την υλοποίηση του χρησιμοποιούμε τις 6 ζητούμενες συναρτήσεις αλλά και μερικές βοηθητικές για την λειτουργία του:

Build Avltree: Δημιουργία δέντρου με εισαγωγή στοιχείων που διαβάζουμε από αρχείο, ένα έναν καλώντας επαναλαμβανόμενα την instert

Get Avl Tree: Επιστρέφει το μέγεθος του δέντρου

Find Min Avl Tree: Βρίσκει και επιστρέφει το ελάχιστο του δέντρου εξερευνώντας κάθε φορά το αριστερό υποδέντρο (όσο αυτό υπάρχει) καθώς για κάθε δυαδικό δέντρο ισχύει (αριστερό παιδί<=δεξί)

Search Avl Tree: Αναζήτηση ενός στοιχείου στο δέντρο και επιστροφή μηνύματος (σωστού/λάθους) ελέγχοντας αν το ζητούμενο στοιχείο είναι κάθε φορά μεγαλύτερο/μικρότερο του κόμβου και επισκέπτοντας αντίστοιχα το ανάλογο υποδέντρο

Insert Avl Tree: Εισαγωγή ενός στοιχείου στο δέντρο διατηρόντας την αύξουσα διάταξη του (φυσικά ως bts έχει σωστή διάταξη με in-order traversal). Αν AVL καινό, δημιουργούμε έναν καινούργιο κόμβο. Αν όχι τότε ανάλογα με στοιχείο του κόμβου τοποθετείτε ως αριστερό ή δεξί παιδί του υπάρχοντα προηγούμενου κόμβου. Έπειτα ελέγχετε η ισορροπία του δέντρου μετρώντας το μέγιστο ύψος. Αν η ισορροπία καταστράφηκε (maxh-minh>1) τότε γίνονται οι κατάλληλες περιστροφές ώστε η ισορροπία του δέντρου να εξισορροπηθεί

Delete Avl Tree: Αρχικά αναζητούμε το στοιχείο που θέλουμε να διαγράψουμε. Έπειτα αφού βρούμε τον κόμβο που αφορά ελέγχουμε

- Αν έχει 2 παιδιά: Βρίσκει αμέσως μικρότερο (successor), πηγαίνουμε στο αριστερό άκρο του δέντρου, αντιγράφουμε τον successor στον κόμβο που επιθυμούμε να διαγράψουμε και τέλος ο successor διαγράφεται πολύ πιο εύκολα και ενημερώνει τους δίκτες current και parent
- Αν έχει 1 παιδί: Τότε απλά κάνουμε τον πατέρα του τωρινού του να δείχνει στο παιδί του τωρινού κόμβου
- Αν δεν έχει παιδιά: Τότε απλά κάνουμε τον πατέρα να δείχνει σε nullptr

Τέλος υπολογίζουμε τα ύψη και αν το δέντρο είναι μη ισορροπημένο καλούμε τις βοηθητικές συναρτήσεις για περιστροφή του και διατήρησή της διάταξης του

! Για κάθε συνάρτηση υπάρχει και η αντίστοιχη βοηθητική της για να καλείτε αναδρομικά μέσα στου υποκόμβους (π.χ. H Get Size Avl Tree- Height)

Πέρα από αυτές οι υπόλοιπες βοηθητικές συναρτήσεις είναι οι:

Get Balance: Υπολογίζει και επιστρέφει διαφορά μεταξύ του αριστερού και του δεξί υποδέντρου καλώντας την συνάρτηση height

Height: Υπολογίζει το ύψος του δέντρου με ρίζα τον κόβμο που δίνεται ως είσοδος επιστρέφοντας το ypsos από ως πεδίο της δομής του κόμβου

Rotate Left: Αριστερόστροφη περιστροφή για να μετατραπη το δέντρο και πάλι σε ιισσοροπιμένο με βάση την θεωρεία (καλείτε όταν το δεξί υποδέντρο είναι μεγαλύτερο του αριστερού κατά ύψος >1)

Rotate Right: Δεξιόστροφη περιστροφή για να μετατραπεί και πάλι το δέντρο σε ισορροπημένο με αλγόριθμο της θεωρίας (καλείτε όταν το ύψος του αριστερού υποδέντρου είναι μεγαλύτερο του δεξί κατά >1)

2iv) Graph- Για ανπαράταση σχέσεων, εύρεσης βέλτιστης διαδρομής (Dijkstra/Prim κλπ)

Υλοποιείτε με έναν δισδιάστατο πίνακα γειτνίασης που που αποθηκεύει το βάρος των συνδέσεων όλων των κόμβων του γράφου(μέγιστης χωρητικότητας 3000 κόμβων) αλλά και μιας μεταβλητής που αποθηκεύει το πλήθος των κόμβων.
Συναρτήσεις:

Build Graph: Δημιουργία γράφου διαβάζοντας στοιχεία από αρχείο και αποθηκεύοντας το βάρος των σχέσεως μεταξύ τους στον πίνακα γειτνίασης. (Αφορώντας φυσικά κατευθυνόμενο Γράφο). Η γραμμές αφορούν τον κόμβο αφετηρίας και οι γραμμές τον κόμβο προορισμού

Get Size Graph: Επιστρέφει το μέγεθος του γράφου δηλαδή το συνολικό πλήθος των κόμβων

Insert Graph: Εισαγωγή μιας ακμής στον γράφου. Αν δεν υπάρχει η αντίστοιχή σύνδεση στον πίνακα τότε προστίθεται και ανανεώνετε το μέγεθος του γράφου

Delete Graph: Διαγραφή μια ακμής από τον γράφο τοποθετώντας μηδενικό βάρος στην αντίστοιχη σχέση στον πίνακα γειτνίασης

Compute Shortest Path: Με τον αλγόριθμο Dijkstra επιλέγουμε ένα κόμβο και θέτουμε όλες τις αποστάσεις αυτόν ένα αριθμό κοντά στο άπειρό. Χρησιμοποιούμε επίσης έναν πίνακα αληθείας για το αν έχουμε επισκευτεί τον κάθε κόμβο. Όσο υπάρχουν κόμβοι που δεν έχουμε επισκευτεί επιλέγουμε τον κόμβο με την μικρότερη απόσταση από τον αρχικό και για κάθε γείτονά του αν η απόσταση από τον αρχικό είναι μικρότερη από αυτή που είναι αποθηκευμένη την ανανεώνουμε. Όταν επισκεφτούμε τον πίνακα dist που περιέχει την ελάχιστη απόσταση από τον αρχικό κόμβο προς όλους του γράφου αλλά εμάς μας αφορά μόνο το τελευταίος τον οποίο και επιστρέφουμε

Compute Spanning Tree: Υπολογισμός MST ελάχιστου δέντρου κάλυψης του γράφου με την βοήθεια του αλγορίθμου Prim. Ξεκινώντας από τον πρώτο κόμβο και επαναλαμβάνοντας μέχρι να επισκεφτούμε όλους τους κόμβους του γράφου υπολογίζουμε την ακμή με το μικρότερο βάρος και την προσθέτουμε στο άθροισμα των ακμών της συντομότερης διαδρομής. Το κάνουμε αυτό με δύο πίνακες, έναν που αποθηκεύει τις ελάχιστες αποστάσεις και έναν αληθείας που αποθηκεύει ποιους κόμβους έχουμε επισκεφτεί.

DFS: Βοηθητική συνάρτηση για αναζήτηση σε βάθος για τον γράφο. Εκτελείτε αναδρομικά και χρησιμοποιείτε και πάλι ένα πίνακας αληθείας για τα στοιχεία που έχουμε επισκεφτεί, χρησιμοποιείτε για την εύρεση των πλήθους των συνδεδεμένων κόμβων

Find Connected Components: Συνάρτηση που υπολογίζει και επιστρέφει το πλήθος των συνεκτικών συνιστωσών του γράφου. Διαδοχικά επισκέπτεται κόμβους που δεν έχει επισκεφτεί (με την βοήθεια του πίνακα αληθείας visited) και εκτελεί την αναδρομική DFS για να μετρήσει τον αριθμών των συνδεδεμένων υπογράφων της του συγκεκριμένου κόμβου

2v) Hashtable- Χρησιμοποιείτε για γρήγορη αναζήτηση σε O(1)

Υλοποιείτε με έναν πίνακα table που αποθηκεύει τον αριθμό των κόμβων στον πίνακα κατακερματισμού και έναν πίνακα αληθείας katilimeno που περιέχει πληροφορία σχετικά με το αν το κελί είναι κατειλημμένο ή όχι. Συναρτήσεις δομής:

Build Hashtbale: Δημιουργία πίνακα κατακερματισμού διαβάζοντας στοιχεία από αρχείο και εισάγοντας τα καλώντας επαναληπτικά την Insert Hashtable

Get Size Hashtable: Επιστρέφει τον αριθμό των στοιχείων που βρίσκονται μέσα στον πίνακα κατακερματισμού, δηλαδή την μεταβλητή size που είναι ιδιότητα της ομώνυμης κλάσης

Insert Hashtable: Εισαγωγή ενός στοιχείου στον πίνακα χρησιμοποιώντας μια γραμμική συνάρτηση κατακερματισμού (num MOD 1000 +i) όπου το i αυξάνετε κάθε φορά που το στοιχείο αδυνατεί να εισαχθεί στοιχείο σε κατειλημμένη θέση. Αυτή είναι μια τεχνική αντιμετώπισης του κλασικού προβλήματος του κατακερματισμού.

Πχ Num1=1003 Num2=4992 Num3=4813

Num	1003	4992	4813
Θέση στον πίνακα	3	2	3(+i)=4

Search Hashtable: Αναζήτηση ενός στοιχείου στον πίνακα ξεκινάει από την δηλομένη θέση με βάση την συνάντηση κατακερματισμού (Num MOD 1000) και ελέγχει σειριακά μέχρι να βρει το στοιχείο και επιστρέφει ανάλογο μήνυμα (SUCCESS-FAILURE)

3. **Output**

Μέσα σε διαδοχικές if που καθορίζουν την εντολή που εκτελείτε χρησιμοποιούμε εκτυπώνουμε το αποτέλεσμα της κάθε εντολής που καλείτε για την αντίστοιχη δομή. Παράλληλα εκτυπώνουμε και τον χρόνο εκτέλεσης της κάθε λειτουργίας με τρόπο που εξηγείτε παρακάτω

Βιβλιοθήκες:

Χρησιμοποιούμε 4 εξωτερικές βιβλιοθήκες

1. Την String για την διαχείρηση των συμβολοσειρών (όπως αυτών που αντλούμε από το αρχείο command.txt)
2. Την fstream για την διαχείρηση των αρχείων τόσο κατά το άνοιγμα (input) όσο και για το αποτέλεσμα της εκτέλεσης (output)
3. Την sstream η οποία μας επιτρέπει να χρησιμοποιούμε strings σαν να ήταν ροές εισόδου
4. Την Chrono για να υπολογίζουμε την ταχύτητα εκτέλεσης κάθε εντολής, κάτι πολύ χρήσιμο για μία βιβλιοθήκη δομών δεδομένων όπου η μελέτη του χρόνου εκτέλεσης και των υπολογιστικών όρων είναι καίρια

Χρονομέτρηση:

- Για την χρονομέτρηση χρησιμοποιούμε την βιβλιοθήκη Chrono για να μετρήσουμε το χρόνο υλοποίησης κάθε εντολής, με την βοήθεια της συνάρτησης high_resolution_clock με την οποοία καταγράφουμε 2 στιγμιότηπα χρόνου και με την χρήση της duration_cast υπολογίζουμε την διαφορά τους (δηλάδη το χρόνο εκτέλεσης της κάθε εντολής) σε nanosecond
- Χρησιμοποιούμε το auto ώστε το αποτέλεσμα των συναρτήσεων της βιβλιοθήκης chrono να αποθηκεύονται σε μεταβλητές που καθορίζονται αυτόματα από τον compiler και να μην δηλώνονται κάθο φορά (πχ start, end)