Αναφορά Παράδοσης

Δομές Δεδομένων 2020-2021

2η εργασία

Φοιτητές:

Ιωάννης Αποστόλου: p3190013

Ηλίας Καλαντζής: p3190068

Java Version:

15.0.1

Ποιος αλγόριθμος ταξινόμησης χρησιμοποιήθηκε;

Ο αλγόριθμος ταξινόμησης που χρησιμοποιήσαμε στην εργασία μας ήταν ο **HeapSort**. Επειδή ήμασταν 2 φοιτητές που είχε ο ένας άρτιο και ο άλλος περιττό τελευταίο ψηφίο του ΑΜ, είχαμε την πολυτέλεια να επιλέξουμε όποιον αλγόριθμο θέλουμε.

Πώς υλοποιήθηκε η Remove;

Η **remove** ζητήθηκε να είναι πολυπλοκότητας **O(LogN)**, όμως εμείς καταφέραμε να βρούμε λύση μόνο για **O(N)** οπότε θα παρουσιάσουμε αυτή. Ως παράμετρο το **id** της πόλης. Υπάρχει η **for** η οποία αυξάνει το **i** κατά ένα κάθε φορά μέχρι να βρεθεί το στοιχείο που θα έχει ίδιο στοιχείο **id** ίδιο με την παράμετρο. Εάν δεν βρεθεί επιστρέφει null, αλλά εαν βρεθεί, θέτει σε μια μεταβλητή **r** το στοχείο που θα επιστρέφει και θα αφαιρεθεί, ενώ παράλληλα θέτει στην θέση που ήταν το στοιχείο μας το τελευταίο στοιχείο, μικραίνει τον πίνακα κατα 1 και μετά τον ταξινομεί και πάλι.

DynamicCovid_k_withPQ

Η βασική ιδέα είναι ότι έχουμε μια μεγιστοστρεφή σωρό η οποία αυξάνεται ανάλογα με το μέγεθος των στοιχείων που προστίθονται μέσα της αλλά φτάνει μέγιστο μέγεθος τον ακέραιο αριθμό **k** που είναι ο αριθμός των πόλεων που θέλουμε να δούμε ταξινομημένες. Κάθε φορά που προστίθεται ένα στοιχείο η ουρά ταξινομείται, ενώ όταν φτάσει **k** μέγεθος το επόμενο που θα πάει να προστεθεί θα το συγκρίνουμε με το μικρότερο και θα εαν είναι μεγαλύτερο θα αφαιρέσουμε το μικρότερο και έπειτα θα προσθέσουμε το στοιχείο, αλλιώς δεν θα προσθέσουμε καν το στοιχείο.

Ο λόγος που ταξινομούμε κάθε φορά την ουρά είναι γιατί θέλουμε να υπάρχει κάτι σαν “live timing” όπου θα βλέπουμε εκείνη την στιγμή στις **x** από τις **y** πόλεις που προστέθηκαν ποια είναι η κατάταξη.

Πιο αναλυτικά στον κώδικα, έχουμε την ουρά τύπου **PQ** στην οποία μετά από κατάλληλους ελέγχους ορθότητας εισέρχονται τα στοιχεία. Στην **PQ** χρησιμοποιούμε την δυνατότητα που μας δίνει η java για πολυμορφισμό, γιαυτό στον κατασκευαστή ανάλογα με το μέρος της άσκησης,

έχουμε διαφορετικό κατασκευαστή και με τις παραμέτρους που παίρνει καταλαβαίνει για το τι να κανει.

Πριν προστεθεί ένα στοιχείο, ελέγχουμε το μέγεθος εάν είναι μικρότερο από το **k** ή όχι. Εαν είναι, τότε προσθέτουμε το στοιχείο και ταξινομείτε στην ουρά. Εάν όχι, τότε συγκρίνουμε το στοιχείο που πάμε να περάσουμε με το μικρότερο και εαν είναι μεγαλύτερο, τότε αφαιρούμε το μικρότερο και βάζουμε το καινούργιο. Αυτά πραγματοποιούνται στην **DynamicCovid_k_withPQ,** καθώς θέλαμε με την ίδια ουρά να λύσουμε ένα πρόβλημα με διαφορετικούς τρόπους.

Η πολυπλοκότητα εισαγωγής (insert) και ταξινόμησης (sort) στο **Μέρος Γ** είναι **O(LogN), O(NLogN)** αντίστοιχα, καθώς χρησιμοποιείται και πάλι **max heap tree**, με την διαφορά ότι ο χώρος που καταλαμβάνει, είναι **k** θέσεις. Αυτό επιτυγχάνεται, με την συνάρτηση **min** όπου σε χρόνο, **O(N/2)** βρίσκει και επιστρέφει το μικρότερο στοιχείο στο τελευταίο επίπεδο του δέντρου.

Η **DynamicCovid_k_withPQ** σίγουρα δεν συμφέρει σε περιπτώσεις όπου το **k** είναι αρκετά μικρό, καθώς θα πρέπει αρκετές φορές να συγκρίνουμε το στοιχείο που έρχεται με το μικρότερο. Επίσης με τον τρόπο που κατασκευάστηκε η **PQ** δεν ξέρουμε κάθε στιγμή το μικρότερο γιαυτό, κάθε φορά πρέπει να το βρίσκουμε από το τελευταίο επίπεδο του δέντρου, μέσω της συνάρτησης **min**.

*Σχόλιο: Στα Μέρη Α, Β και Γ χρησιμοποιήσαμε **μόνο** δομές **μεγισοστραφού σωρού (**max heap tree). Καθώς έτσι αναφέρεται στο **hint**. Κάποιες πολυπλοκότητες θα μπορούσαν να είχαν υλοποιηθεί σε καλύτερο χρόνο, με την χρήση ελαχισοστραφού σωρού αλλά θα χρειαζόμασταν μεγαλύτερο όγκο μνήμης.*

Bonus

Η υλοποίηση **Dynamic Median** έγινε με την χρήση 2 δομών τύπου **max heap tree** και **min heap tree.** Τα οποία αποθήκευαν στο εσωτερικό τους, μέρος των δεδομένων (ισάριθμο και τα 2 μεταξύ τους). Με τον τρόπο αυτό, είχαμε πρόσβαση οποιαδήποτε στιγμή, στο **median** σε χρόνο **O(1).** Η εισαγωγή (insert), των στοιχείων γινόταν σε χρόνο **O(LogN).** Αυτός ο τρόπος προσέγγισης λειτουγεί με την εξής λογική: Με την εισαγωγή ενώς νέου στοιχείου, ελέγχεται αν είναι μεγαλύτερο ή μικρότερο του ήδη υπάρχον median και όταν πρόκειται για μεγαλύτερο, εισάγεται στην **max heap tree,** αλλιώς στην **min heap tree** και μετέπειτα τα 2 δέντρα επαναισορροπουνται με την rebalance. Τέλος στην περίπτωση, όπου τα 2 δέντρα έχουν το ίδιο μέγεθος (δηλαδή έχουμε ζυγό αριθμό στοιχείων, η δομή επιστρέφει το στοιχείο με το μεγαλύτερο ποσοστό κρουσμάτων ανα 50.000).

Οδηγίες

Εάν υπάρχει κάποιο πρόβλημα κατά το τρέξιμο του προγράμματος μπορείτε να δείτε τα παρακάτω:

- Δείτε εάν έχετε εγκατεστημένη την **Java 15.0.1**
- Ο κώδικας μας είναι σε πακέτο, ώστε να επικοινωνούν εύκολα τα αρχεία μας, άρα πρέπει να γίνουν compile κατά αυτόν τον τρόπο
- Το πρόγραμμα είναι σε πακέτα οπότε δοκιμάστε να κάνετε compile με "*javac -d . com/oneprog/*.java*" και "*javac -d . com/exceptions/*.java*"
- Έπειτα θα δημιουργηθούνε τα αρχεία .class μέσα στον ίδιο φάκελο ή σε διαφορετικό ανάλογα με το path που θα έχετε.
- Τρέξετε την εντολή "*java oneprog/Covid_k.java*" για το **Μέρος Α,** την "*java oneprog/PartB*" για το **Μερος Β** (για debugging λόγους) και την "*java oneprog/DynamicCovid_k_withPQ*" για το **Μέρος Γ**. Για να τρέξετε ένα ένα τα προγράμματα ανά μέρος.

IDE: Intellij (της JetBrain).

Η **είσοδος** των δεδομένων πραγματοποιείται από ένα αρχείο το οποίο πρέπει να τοποθετηθεί στον φάκεο **"text_file".** Το αρχείο πρέπει να είναι τύπου **txt**, ενώ όταν ο χρήστης θα εκτελέσει το πρόγραμμα το πρώτο πράγμα που θα ζητήσει το πρόγραμμα θα είναι το όνομα του αρχείου. Το πρόγραμμα ελένγχει εάν βρίσκει το αρχείο, οπότε εαν γράψετε λάθος το όνομα θα σας το ξαναζητήσει.

Για οποιοδήποτε θέμα ή απορία μπορείτε να επικοινωνήσετε στα ακαδημαϊκά μας email, p3190013@aueb.gr [Ιωάννης Αποστόλου] και p3190068@aueb.gr [Ηλίας Καλαντζής]