Μέρος Ε – Αναφορά

Τανέρ Ιμάμ p3200057

Ιπποκράτης Παντελίδης p3210150

1) Μέρος Α

Στο μέρος Α φτιάξαμε την κλάση MaxPQImpl που υλοποιεί χρησιμοποιώντας generics το interface MaxPQ.Ειδικότερα φτιάξαμε μια ουρά προτεραιότητας χρησιμοποιώντας σωρό. Η ουρά έχει έναν Constructor μέσα από τον οποίο παίρνει το μέγεθος της και στον οποίο δηλώνεται ο Comparator δηλαδή ο τρόπος με τον οποίο θα κάνει τις συγκρίσεις μεταξύ των κελιών του σωρού και έχει επίσης τις μεθόδους:

- insert(): η οποία προσθέτει ένα νέο στοιχείο στον σωρό εκτός αν δεν χωράει, όπου τότε αυξάνει το μέγεθος του μέσω της grow(),και μετά το αναδύει στην κατάλληλη θέση μέσω της swim().
- getMax():η οποία παίρνει το μεγαλύτερο στοιχείο του σωρού, το οποίο βρίσκεται στην ρίζα και το ανταλλάζει με το τελευταίο. Το αφαιρεί και το επιστρέφει και έπειτα καταδύει την νέα ρίζα μέσω της sink() στην κατάλληλη θέση.
- peek():επιστρέφει χωρίς να αφαιρεί την ρίζα, δηλαδή το μεγαλύτερο στοιχείο του σωρού.
- swap():αλλάζει την θέση δύο στοιχείων του σωρού.

2) Μέρος Β

Στο μέρος Β έχουμε φτιάξει αρχικά μία μέθοδο read() που διαβάζει ένα αρχείο, όπου πρώτα μετράει τις γραμμές του για να δημιουργήσει ένα πίνακα τέτοιου μεγέθους. Έπειτα το ξαναδιαβάζει και περνάει τα δεδομένα στον πίνακα εφόσον ισχύουν οι συνθήκες αλλιώς τερματίζει το πρόγραμμα και τέλος επιστρέφει τον πίνακα. Στην συνέχεια έχουμε υλοποιήσει την μέθοδο Allocation() που παίρνει σαν όρισμα έναν πίνακα φακέλων και τους τοποθετεί σε δίσκους. Αρχικά δημιουργούμε τον πρώτο δίσκο και αρχικοποιούμε με 0 μεταβλητές για τα

στοιχεία του πίνακα, για το άθροισμα των φακέλων και το πλήθος τους. Φτιάχνουμε επίσης την ουρά προτεραιότητας με generics ώστε να παίρνει δίσκους στην οποία προσθέτουμε τον πρώτο δίσκο. Παίρνουμε ένα loop όσο το μέγεθος του πίνακα και κάθε φορά μέσω της peek() και της χρήσης της Disk Comparator έχουμε τον δίσκο με τον μεγαλύτερο ελεύθερο χώρο. Βλέπουμε αν αυτός είναι μεγαλύτερος από τον αντίστοιχο φάκελο και κάνουμε ορισμένες τροποποιήσεις. Αν όμως δεν χωράει δημιουργούμε νέο δίσκο με αυξημένο id και κάνουμε τις ίδιες αλλαγές με πάνω. Όταν γίνουν οι τροποποιήσεις αφαιρούμε το max μέσω της getMax() και το ξαναπροσθέτουμε μέσω της insert() ώστε να ανανεωθούν οι αλλαγές. Τέλος, αν το πλήθος των φακέλων είναι μικρότερο των 100 εκτυπώνουμε τους δίσκους με φθίνουσα σειρά ελεύθερου χώρου χρησιμοποιώντας επαναληπτικά και όσο το id την getMax(). Επομένως η χρήση της ουράς προτεραιότητας σε αυτόν τον αλγόριθμο είναι να έχει αποθηκευμένους τους δίσκους, αλλά κυρίως το ότι η θέση τους στην ουρά εξαρτάται από ένα κλειδί, που στον αλγόριθμο μας είναι ο ελεύθερος χώρος. Έτσι μπορούμε να βρίσκουμε πολύ εύκολα τον δίσκο με τον περισσότερο ελεύθερο χώρο που τον χρειαζόμαστε συνέχεια, χωρίς προσπέραση όλων των στοιχείων, απλά χρησιμοποιώντας την μέθοδο peek().

3) Μέρος Γ

Στο μέρος Γ χρησιμοποιούμε τον αλγόριθμο ταξινόμησης MergeSort για να ταξινομήσουμε σε φθίνουσα σειρά τον πίνακα που προκύπτει από την μέθοδο read() του ερωτήματος Β. Ειδικότερα βλέπουμε αν ο πίνακας έχει παραπάνω από ένα στοιχεία, αν ναι χωρίζουμε το διάστημα στην μέση και κάνουμε αναδρομικά χωριστή ταξινόμηση σε κάθε υποπίνακα. Κάθε φορά μετά την ταξινόμηση συγχωνεύουμε τους ταξινομημένους υποπίνακες με την μέθοδο merge() και στο τέλος έχουμε τον πίνακα μας ταξινομημένο με φθίνουσα σειρά.

4) Μέρος Δ

Στο Μέρος Δ έχουμε την κλάση Evaluate Algorithms που συγκρίνει τους δύο αλγορίθμους με βάση τυχαίες τιμές. Αρχικά υλοποιήσαμε την μέθοδο writetoFile() με την οποία μπορούμε να γράψουμε σε ένα αρχείο. Έπειτα έχουμε

την μέθοδο RandomIn() η οποία χρησιμοποιώντας την Random της java αρχικοποιεί τιμές σε αρχεία. Ειδικότερα παράγει 10 αρχεία για τιμές του N (100, 500, 1000) όπου N είναι οι φάκελοι που δίνονται τυχαία. Έτσι δημιουργούμε 10 αρχεία για N=100, άλλα 10 για N=500 και τέλος άλλα 10 για N= 1000. Μετά στην main καλούμε την μέθοδο RandomIn() για να δημιουργηθούν τα αρχεία , με τις τυχαίες τιμές και μετά φτιάχνουμε μεταβλητές, οι οποίες αρχικά θα αποθηκεύονται με 0, και θα μετρούν το σύνολο των δίσκων που χρειάζεται η κάθε κατηγορία. Για να διαβάσουμε τα αρχεία κάθε κατηγορίας και να τρέξουμε τους αλγορίθμους φτιάχνουμε ένα loop από 0 μέχρι 9 και για τον πρώτο αλγόριθμο της πρώτης κατηγορίας(N=100) διαβάζουμε επαναληπτικά τα αρχεία μέσω της μεθόδου read() της κλάσης Greedy, έπειτα μοιράζουμε τους φακέλους στους δίσκους μέσω της Allocation() πάλι της Greedy και ανανεώνουμε την κατάλληλη μεταβλητή με την τιμή που επιστρέφει η Allocation(), δηλαδή τον πλήθος των απαιτούμενων δίσκων. Για τον δεύτερο αλγόριθμο καλούμε όπως ακριβώς και για τα ίδια αρχεία στον πρώτο την read() της Greedy, τον πίνακα που αυτή επιστρέφει τον ταξινομούμε κατά φθίνουσα σειρά μέσω της μεθόδου MergeSort() της κλάσης Sort και μετά παρομοίως καλούμε την Allocation() για να τοποθετήσει τους φακέλους στους δίσκους και ανανεώνουμε πάλι την μεταβλητή. Κάνουμε ακριβώς την ίδια δουλειά και για τις υπόλοιπες κατηγορίες και αφού ολοκληρώσουμε τον διάβασμα των αρχείων οι μεταβλητές μας θα έχουν πάρει τις επιθυμητές τιμές. Διαιρούμε την κάθε μεταβλητή με το 10 για να βρούμε τον μέσο όρο και είμαστε έτοιμοι για συγκρίσεις. Βλέπουμε για κάθε κατηγορία ποιος μέσος όρος είναι μικρότερος, αυξάνουμε ένα μετρητή και εμφανίζουμε το κατάλληλο μήνυμα. Ο αλγόριθμος με τον μεγαλύτερο μετρητή είναι ο αποδοτικότερος. Στο παράδειγμα μας παρατηρούμε ότι αποδοτικότερος αλγόριθμος είναι ο $2^{ος}$. Παρατηρούμε επίσης ότι στην πρώτη κατηγορία η διαφορά τους είναι σχετικά μικρή (5-10 δίσκοι), όσο όμως αυξάνεται το N αυξάνεται και η διαφορά τους. Για N=500 έχουν διαφορά περίπου 30 δίσκων ενώ για N=1000 περίπου 80 δίσκων.

Παραδείγματα Μέσων Όρων για μια τυχαία αρχικοποίηση των αρχείων εισόδου:

$1^{ος}$ Αλγόριθμος:

N = 100 : Average_N1_100 = 61

N = 500 : Average_N1_500 = 295

N = 1000 : Average_N1_1000 = 582

2ος Αλγόριθμος:

N = 100 : Average_N2_100 = 55

N = 500 : Average_N2_500 = 257

N = 1000 : Average_N2_1000 = 505