**Μαρία Σχοινάκη**

Τα αρχεία για το Μέρος Δ είναι τα εξής : **MerosDcompare.java, MerosDcreate.java, MerosDgreedydecreasing.java . Επιπλέον έχουμε δημιουργήσει μια δεύτερη MaxPQ(IntMaxPQ) η οποία χειρίζεται int αντί για Disk για την υλοποίηση της Heapsort στο Μέρος Γ ( Ταξινόμηση φακέλων).**

**Μέρος Α – Disk, MaxPQ**

**ΑΡΧΕΙΑ**

**Disk.java, MaxPQ.java**

Στην εργασία ξεκινήσαμε υλοποιώντας την κλάση **Disk**, η οποία παριστάνει δίσκο, με μοναδικό id, λίστα με τους φακέλους του τρέχοντος δίσκου και μέθοδο με το πόσο ελεύθερο χώρο έχει ο τρέχον δίσκος. Στην συνέχεια υλοποιήσαμε την ουρά προτεραιότητας για τα αντικείμενα τύπου Disk, την κλάση **MaxPQ** (Ίδια υλοποίηση με του φροντιστηρίου). Ορίσαμε έναν πίνακα τύπου Disk και μια μεταβλητή που μετράει το πλήθος των δίσκων στην ουρά. Υλοποιήσαμε την μέθοδο **add()** που προσθέτει ένα στοιχείο(*δίσκο*) στην ουρά, και αν είναι το τελευταίο στοιχείο που χωράει, καλεί την **grow()**, που πρακτικά φτιάχνει μεγαλύτερου μεγέθους πίνακα, βάζει τα στοιχεία του παλιού μέσα και έπειτα η **insert()**, προσθέτει το προς προσθήκη στοιχείο. Για να γίνει όμως η υλοποίηση της **insert()**, χρειάζεται και η **swim()**, η οποία κάνει τις κατάλληλες αλλαγές ώστε να ισχύει η ιδιότητα του σωρού. (*Το κλειδί ενός κόμβου πρέπει να είναι μικρότερο από αυτό του γονέα του*). Ανάλογα υλοποιήσαμε και την μέθοδο **getmax()**, η οποία αφού ελέγχει ότι υπάρχει στοιχείο προς εξαγωγή, εξάγει το μεγαλύτερο και καλεί την μέθοδο **sink()**, η οποία κάνει τις κατάλληλες αλλαγές ώστε να ισχύει η ιδιότητα του σωρού. (*Το κλειδί ενός κόμβου πρέπει να είναι μικρότερο από αυτό του γονέα του*). Για την υλοποίηση της **swim()** και της **sink()**, υλοποιήσαμε και την μέθοδο **swap()**, η οποία αντιμεταθέτει 2 κόμβους μεταξύ τους. Τέλος, υλοποιήσαμε τις μεθόδους **peek()** και **getSize()**, οι οποίες επιστρέφουν το στοιχείο προς εξαγωγή(*μεγαλύτερη προτεραιότητα, αν αυτό υπάρχει*) και το πλήθος των δίσκων στην ουρά, ανάλογα.

**Μέρος Β – Αλγόριθμος 1**

**ΑΡΧΕΙΑ**

**Greedy.java, MaxPQ.java, Disk.java, List.java, Node.java**

Σε αυτό το σημείο, ξεκινήσαμε υλοποιώντας την κλάση Greedy. Πρακτικά, η κλάση διαβάζει ένα αρχείο με ακεραίους(*μεγέθη φακέλων σε MB*), τους βάζει σε μία λίστα, και χωρίς να

αλλάξει την σειρά τους, τους προσθέτει σε κάθε δίσκο ανάλογα. Η επιλογή τώρα, του δίσκου εισαγωγής γίνεται με την ουρά προτεραιότητας MaxPQ. Θεωρείται ως μεγαλύτερο στοιχείο, ο δίσκος με την μεγαλύτερη ελεύθερη χωρητικότητα και ανάλογα αν χωράει ο φάκελος μπαίνει σε αυτόν ή στον επόμενο μεγαλύτερο κοκ. Προφανώς, κάθε φορά που εισάγεται ένας φάκελος στον δίσκο με την μεγαλύτερη χωρητικότητα που παράλληλα χωράει και τον φάκελο, αλλάζει η σειρά της ουράς προτεραιότητας(*αν χρειάζεται*), γιατί πλέον ο συγκεκριμένος δίσκος δεν έχει την ίδια ελεύθερη χωρητικότητα και πιθανόν να μην είναι πλέον ο max. Η Greedy επίσης, περιέχει το διάβασμα του αρχείου με τους φακέλους και ελέγχει αν κάθε φάκελος είναι χωρητικότητας μεταξύ 0 και 1.000.000(*αλλιώς εμφανίζει μήνυμα λάθους*). Τέλος, αν το πλήθος των φακέλων ξεπερνάει το 100, τότε εκτυπώνεται το πλήθος των δίσκων που χρησιμοποιήθηκαν και το συνολικό άθροισμα σε TB των φακέλων. Σε άλλη περίπτωση εμφανίζεται το πλήθος των δίσκων που χρησιμοποιήθηκαν, το συνολικό άθροισμα σε TB των φακέλων **και** το περιεχόμενο των δίσκων ανάλογα με το ποιος δίσκος έχει την μεγαλύτερη ελεύθερη χωρητικότητα.

**Μέρος Γ – Αλγόριθμος 2**

**ΑΡΧΕΙΑ**

**Sort.java, MaxIntPQ.java, List.java, Node.java**

Ο αλγόριθμος heapsort αρχικά δημιουργεί ένα σωρό (int) στον οποίο εισάγονται οι φάκελοι. Στη συνέχεια επαναλαμβάνει το εξής: αφαιρεί το μεγαλύτερο στοιχείο (*της ρίζας του σωρού*) και το εισάγει στο τέλος της λίστας. Έτσι η λίστα καταλήγει να είναι ταξινομημένη σε φθίνουσα σειρά. Στο κομμάτι αυτό της εργασίας μας, χρησιμοποιήθηκε η heapsort, ώστε να μπορούν να διαταχθούν σε σειρά φθίνουσα οι φάκελοι που περιέχονται στο αρχείο εισόδου, έτσι ώστε να γίνει ένα πείραμα μέσα στα πλαίσια βελτίωσης του αλγορίθμου 1, με στόχο την εξοικονόμηση δίσκων. Που τελικά και ισχύει. Αν οι φάκελοι είναι διατεταγμένοι, με σειρά φθίνουσα τότε χρησιμοποιούνται λιγότεροι δίσκοι σε σχέση με τον αλγόριθμο 1, που οι φάκελοι ήταν με την σειρά που διαβάστηκαν(*αταξινόμητοι*).

**Μέρος Δ – Πειραματική σύγκριση**

**ΑΡΧΕΙΑ**

**Greedy.java, MaxPQ.java, Disk.java, List.java, Node.java, Sort.java, MaxIntPQ.java, MerosDcompare.java, MerosDcreate.java, MerosDgreedydecreasing.java**

**Μαρία Σχοινάκη**

Αρχικά, δημιουργήσαμε στον φάκελο που περιέχει τα .java αρχεία μας 3 φακέλους, έναν για κάθε πλήθος φακέλων(*100, 500, 1000*). Κάθε φάκελος θα πρέπει να περιέχει 10 txt αρχεία που θα περιέχουν ανάλογα το πλήθος των φακέλων τόσους τυχαίους αριθμούς στο διάστημα [0, 1000000]. Δηλαδή ο φάκελος n500 περιέχει 10 txt αρχεία, που στο καθένα περιέχονται 500 τυχαίοι αριθμοί στο [0, 1000000]. Αυτό υλοποιήθηκε μέσω της κλάσης MerosDcreate. Ξεκινήσαμε αρχικοποιώντας μια μεταβλητή με το πλήθος των εκάστοτε φακέλων(*100 ή 500 ή 1000*), μία με το όνομα του φακέλου που θέλουμε να μπει το txt αρχείο που θα δημιουργηθεί(*n100, n500, n1000*) και μια με το όνομα "input" που είναι το όνομα κάθε txt αρχείου, συνοδευόμενο από έναν αριθμό(*1, 2,…,10*). Τρέξαμε μια επανάληψη 10 φορές και σε κάθε επανάληψη φτιάχναμε ένα txt αρχείο. Πρώτα αρχικοποιήσαμε τον FileWriter με το να του ορίσουμε το string που περιέχει το όνομα του φακέλου εισαγωγής καθώς και το όνομα που θέλουμε να έχει το txt αρχείο(*input1…*). Έπειτα τρέξαμε μία εσωτερική επανάληψη, η οποία «έγραφε» κάθε αριθμό με τον writer. Προφανώς η επανάληψη έτρεχε σύμφωνα με την μεταβλητή του πλήθους των φακέλων που είχαμε ήδη αρχικοποιήσει. Για να ορίσουμε τώρα τον τυχαίο αριθμό, χρησιμοποιήσαμε την μέθοδο random η οποία δίνει έναν τυχαίο αριθμό μεταξύ 0 και 1. Οπότε πολλαπλασιάσαμε το αποτέλεσμα της random με το 1000000 για να έχουμε το επιθυμητό αποτέλεσμα. Μετά, βάλαμε μια if η οποία έλεγχε ποια γραμμή του txt αρχείου γράφαμε. Αν γράφαμε την τελευταία έγραφε απλά τον αριθμό, σε κάθε άλλη περίπτωση γράφαμε τον αριθμό συνοδευόμενο με έναν χαρακτήρα αλλαγής γραμμής, για να κατέβει στην επόμενη. Τέλος, κλείσαμε το αρχείο. Στην περίπτωση που κάτι πάει λάθος και δεν μπορέσουν να δημιουργηθούν τα αρχεία η try-catch εμφανίζει μήνυμα λάθους. Αξίζει να σημειωθεί ότι το αποτέλεσμα της random*1000000 το μετατρέψαμε σε int γιατί by default ήταν double και όταν το γράφαμε το κάναμε string. Τρέξαμε 3 φορές την MerosDcreate, με διαφορετικά δεδομένα κάθε φορά(*Πλήθος φακέλων 100, όνομα φακέλου που περιέχει τα txt αρχεία n100 - πλήθος φακέλων 500, όνομα φακέλου που περιέχει τα txt αρχεία n500 και αντίστοιχα για 1000*) και δημιουργήσαμε τα 30 txt αρχεία, 10 για κάθε φάκελο(*n100, n500, n1000*).

Για να τρέξουμε τους 2 αλγορίθμους και μετέπειτα να τους συγκρίνουμε, δημιουργήσαμε την κλάση MerosDcompare, αλλά και την κλάση MerosDgreedydecreasing, η οποία χρησίμευσε για να τρέξουμε τον αλγόριθμο 2 σαν main class (Ίδια με την Greedy με τη μόνη διαφορά ότι καλεί τη μέθοδο sort στη λίστα των φακέλων). Αρχικά δημιουργήσαμε έναν πίνακα με string, στον οποίο εισάγεται το path των εκάστοτε 10 txt αρχείων που θέλουμε να τρέξουμε. Κάθε φορά βέβαια αλλάζει το όνομα του φακέλου(*n100, n500, n1000*) δηλαδή πρέπει να τρέξουμε τον κώδικα για κάθε ένα από τους τρεις φακέλους(*n100, n500, n1000*). Έπειτα τρέξαμε μια επανάληψη 10 φορές, η οποία τρέχει πρακτικά κάθε αρχείο στην greedy και στην greedydecreasing. Πρακτικά με την εντολή Greedy.main(arg), καλούμε την main της greedy και της δίνουμε ως όρισμα τον πίνακα με τα ονόματα των txt αρχείων που δημιουργήσαμε. Μετά η greedy χάρις την static μεταβλητή iter, έχει πρόσβαση σε κάθε στοιχείο του πίνακα, διαφορετικό κάθε φορά(αφού είναι static άρα διατηρείται η τιμή της, η οποία αυξάνεται κατά 1 πριν το τέλος κάθε εκτέλεσης). Το ίδιο ισχύει και για την greedydecreasing. Έτσι, τρέχουμε τους 2 αλγορίθμους οι οποίοι εκτυπώνουν ο καθένας

ξεχωριστά το άθροισμα των φακέλων και το πλήθος των δίσκων που χρησιμοποιήθηκαν, για κάθε αρχείο ξεχωριστά. Τέλος, η MerosDcompare εμφανίζει το συνολικό μέγεθος όλων των φακέλων που χρησιμοποιήθηκαν(*Σε κάθε 10αδα txt αρχείων*), το σύνολο των δίσκων που χρησιμοποίησε η greedy αλλά και η greedydecreasing, και τους αντίστοιχους μέσους όρους. Όλα αυτά τα μεγέθη υπάρχουν λόγω ανάλογων στατικών μεταβλητών που χρησιμοποιήθηκαν στους 2 αλγορίθμους. Οι στατικές μεταβλητές παραμένουν ίδιες κάθε φορά που τρέχει η κλάση, δηλαδή δεν αρχικοποιούνται ξανά, έχουν ως αρχική τιμή την τιμή που είχαν την τελευταία φορά που έτρεξε η κλάση.(*πχ 1$^{ο}$ run κάνουμε το static n = 0, n += 2. Στο 2$^{ο}$ run θα είναι n = 2 και μετά n = 4*). Αξίζει να σημειωθεί πως η μεταβλητή που μετράει το συνολικό μέγεθος των φακέλων είναι τύπου long γιατί πιθανόν να βγούμε εκτός ορίων του τύπου int. Επίσης για να εμφανίσουμε τα μεγέθη σε TB διαιρούμε με το 1000000 και μάλιστα κάνουμε πρώτα type casting σε double για να μην έχουμε ακέραια διαίρεση, αλλά την κανονική. Το ίδιο ισχύει και για τους μέσους όρους.

Για 100 φακέλους

```
--------------- Final Reports ---------------
Size of all folders : 502.055471TB
Greedy.java used: 591 disks
Sort.java(Greedy-decreasing) used: 527 disks
M.O. Diskwn Greedy: 59.1
M.O. Diskwn Sort(Greedy-Decreasing): 52.7
```

Για 500 φακέλους

```
--------------- Final Reports ---------------
Size of all folders : 2509.801839TB
Greedy.java used: 2953 disks
Sort.java(Greedy-decreasing) used: 2580 disks
M.O. Diskwn Greedy: 295.3
M.O. Diskwn Sort(Greedy-Decreasing): 258.0
```

Για 1000 φακέλους

```
--------------- Final Reports ---------------
Size of all folders : 5052.404286TB
Greedy.java used: 5933 disks
Sort.java(Greedy-decreasing) used: 5146 disks
M.O. Diskwn Greedy: 593.3
M.O. Diskwn Sort(Greedy-Decreasing): 514.6
```

**Μαρία Σχοινάκη**

Παραπάνω φαίνονται τα αποτελέσματα των 2 αλγορίθμων για τα αρχεία που δημιουργήσαμε. Όπως βλέπουμε, για 100 φακέλους, η greedy, χρησιμοποίησε κατά μέσο όρο 59,1 δίσκους ανά txt αρχείο, ενώ η greedydecreasing, 52,7. Δηλαδή η greedydecreasing χρησιμοποίησε 7 λιγότερους δίσκους, κατά μέσο όρο από την greedy, για να κάνει την ίδια δουλειά. Για 500 φακέλους, η greedy, χρησιμοποίησε κατά μέσο όρο 295,3 δίσκους ανά txt αρχείο, ενώ η greedydecreasing, 258,0. Δηλαδή η greedydecreasing χρησιμοποίησε 38 λιγότερους δίσκους, κατά μέσο όρο από την greedy. Τέλος, για 1000 φακέλους, η greedy, χρησιμοποίησε κατά μέσο όρο 593,3 δίσκους ανά txt αρχείο, ενώ η greedydecreasing, 514,6. Δηλαδή η greedydecreasing χρησιμοποίησε 79 λιγότερους δίσκους, κατά μέσο όρο από την greedy. Άρα ενώ όλες οι τιμές ήταν τυχαίες, φαίνεται ξεκάθαρα ότι η greedydecreasing χρησιμοποιεί λιγότερους δίσκους για να κάνει την δουλειά. Θεωρητικά η διαφορά, φαίνεται μικρή, στην πραγματικότητα όμως κάθε δίσκος κοστίζει αρκετά, οπότε είναι σχεδόν αναγκαία η προσπάθεια εύρεσης του βέλτιστου τρόπου. Δεν έχει βρεθεί ακόμα ο αποδοτικότερος αλγόριθμος για το συγκεκριμένο πρόβλημα, δεν μπορεί να αμφισβητήσει κανείς όμως, ότι με την χρήση της ταξινόμησης στο ΄γ ερώτημα βελτιώσαμε σε μεγάλο βαθμό την απόδοση.