ΕΡΓΑΣΙΑ 2

Δομές Δεδομένων

Ονοματεπώνυμο: Κεχριώτη Ελένη

2022-2023

Μέρος Α

Για την υλοποίηση της ουράς προτεραιότητας πήραμε και προσαρμόσαμε την ουρά προτεραιότητας του φροντιστήριου. Η κλάση *MaxPQ* περιέχει όλες τις γνωστές μεθόδους μια ουράς, δηλαδή την *insert*, την *getMax*, την *peek* και την *getSize*, αλλά και κάποιες private μεθόδους για πιο εύκολη υλοποίηση των προηγούμενων μεθόδων, όπως τη *grow*, *sink*, *swim* και *swap*.
Στην κλάση MaxPQ έχουμε έναν σωρό και μια ακέραια μεταβλητή size που συμβολίζει το μέγεθος του σωρού. Η στατική μεταβλητή default_capacity είναι το αρχικό μέγεθος του σωρού και η στατική μεταβλητή autogrow είναι το μέγεθος της αύξησης του σωρού όταν χρειαστεί. Στον constructor δημιουργείται νέος πίνακας τύπου Disk μεγέθους default_capacity και θέτουμε ως size την τιμή μηδέν.

Σύντομη περιγραφή Μεθόδων

Insert(item): Εισάγουμε νέο Δίσκο στον σωρό. Αν ο σωρός είναι γεμάτος καλείται η συνάρτηση grow() για να αυξηθεί το μέγεθος του και αποθηκεύουμε το στοιχείο στην αμέσως επόμενη ελεύθερη θέση. Έπειτα, καλείται η swim() για να αποκατασταθεί η προτεραιότητα.

Peek(): Αν ο σωρός είναι άδειος επιστρέφει null , αλλιώς επιστρέφει το στοιχείο της πρώτης θέσης του, δηλαδή το στοιχείο με το μέγιστο κλειδί .

getMax(): Κάνει ακριβώς ότι και η peek() με την διαφορά ότι αφαιρεί το στοιχείο της πρώτης θέσης, μειώνει το μέγεθος του σωρού και αναδιατάσσει τα υπολειπόμενα στοιχεία του, με τη βοήθεια της sink().

Swim(int i): Στη συνάρτηση αυτή όσο ο γονέας του i είναι έχει λιγότερο ελεύθερο χώρο από ότι το παιδί i, ανταλλάσσεται η θέση τους στο σωρό. Αν i=1 σημαίνει ότι το i δεν έχει γονέα αρα τελειώνει εκεί η χρησιμότητα της συνάρτησης .

Sink(int i): Στη συνάρτηση αυτή ελέγχει αν το i έχει παιδιά (αν το 2*i,2*i+1 είναι εντός του μεγέθους size) έπειτα συγκρίνει τα παιδιά βρίσκοντας το μεγαλύτερο (max) και μετά συγκρίνει τον γονέα i με το max. Αν το max παιδί έχει περισσότερο ελεύθερο χώρο από ότι ο γονέας ανταλλάσσεται η θέση τους στο σωρό και ως γονέας θεωρείται το (πρώην) παιδί και κάνουμε το ίδιο πράγμα για τα παιδιά του.Η επανάληψη τελειώνει όταν φτάσουμε στα φύλλα του σωρού.

Swap(int i,int j): Ανταλλάσσει τις θέσεις στον σωρό των δυο στοιχείων μέσω παρένθετης προσωρινής μεταβλητής.

Grow(): Δημιουργείται νέο αντικείμενο σωρού μεγέθους size+autogrow. Μέχρι το στοιχείο νούμερο size αντιγράφονται τα στοιχεία του προηγούμενου σωρού.

Μέρος Β

Αρχικά, καλούμαστε να διασχίσουμε ένα αρχείο txt του οποίου το περιεχόμενο είναι τα μεγέθη σε MB, φακέλων που πρέπει να αποθηκευτούν σε δίσκους. Η διάσχιση αυτή γίνεται στη μέθοδο *readFile* της κλάσης **Greedy**. Για την δική μας διευκόλυνση αποθηκεύσαμε τα δεδομένα του αρχείου σε ένα πίνακα ακεραίων, folders.
Στον αλγόριθμο 1 χρησιμοποιούμε την ουρά προτεραιότητας, ως μια πιο αποτελεσματική δομή για την ευκολότερη υλοποίηση του αλγορίθμου. Αρχικά, δημιουργούμε μια ουρά με όνομα disks και όρισμα το DiskComparator.

Ο *DiskComparator* είναι ένας συγκριτής, ο οποίος συγκρίνει δύο αντικείμενα Disks με βάση τον ελεύθερό τους χώρο. (Η κλάση DiskComparator υλοποιεί τη διεπαφή Comparator)
Έπειτα, με τη μέθοδο *insert()*, ωθούμε στη ουρά ένα νέο κενό Δίσκο στον οποίο θα αποθηκεύσουμε ύστερα τους φακέλους.
Με μια επανάληψη for τρέχουμε τον παρακάτω αλγόριθμο για όλους τους φακέλους.
Αν ο φάκελος χωράει στον δίσκο, τότε προσθέτουμε στον δίσκο τον τρέχων φάκελο, διαφορετικά δημιουργούμε έναν νέο Δίσκο στον οποίο θα προσθέσουμε τον φάκελο και τον ωθούμε στην ουρά disks.
Η κύρια χρήση της ουράς προτεραιότητας φαίνεται από τις γραμμές 51-54 στον κώδικα.
Με την μέθοδο peek() βρίσκουμε τον πρώτο δίσκο της ουράς, ο οποίος θα έχει και τον μεγαλύτερο ελεύθερο χώρο. Οπότε αν ο ελεύθερος χώρος του πρώτου δίσκου (*disks.peek().getFreeSpace()*) επαρκεί για την αποθήκευση του φακέλου, τότε μετά από αυτήν οφείλουμε να ενημερώσουμε την προτεραιότητα της ουράς. Στην μεταβλητή τυπου Disk d αποθηκεύουμε τον πρώτο δίσκο της ουράς, τον δίσκο τον οποίο μόλις επεξεργαστήκαμε. Χρησιμοποιούμε την μέθοδο *getMax()* για να αφαιρέσουμε από τη ουρά το πρώτο της στοιχείο, το δίσκο που μέχρι στιγμής είχε την μεγαλύτερη προτεραιότητα (τον μεγαλύτερο ελεύθερο χώρο). Ύστερα, ξαναωθούμε με την insert*()* τον δίσκο, έτσι ώστε να βρει τώρα την νέα θέση του στη ουρά, αφού του αλλάξαμε την προτεραιότητα.
Ωστόσο, αν ο φάκελος δεν χωράει στον πρώτο δίσκο, ο οποίος έχει τον μεγαλύτερο ελεύθερο χώρο, τότε δεν θα χωράει σε κανέναν από τους υπόλοιπους δίσκους της ουράς. Επομένως, θα πρέπει να κατασκευάσουμε ένα νέο δίσκο, στο οποίο θα αποθηκευτεί ο φάκελος και θα ωθηθεί στην ουρά.

Μέρος Γ

Στην κλάση sort υλοποιούμε την μέθοδο ταξινόμησης *quicksort*, η οποία ταξινομεί έναν πίνακα κατά φθίνουσα σειρά.

Μεθόδοι:

Quicksort: Αν το αρχικό σημείο είναι μικρότερο από ότι το τελικό καλείται η μέθοδος partition στην οποία όσα στοιχεία του πίνακα είναι μεγαλύτερα από το στοιχείο στη θέση end τοποθετούνται αριστερά του και οσα είναι μικρότερα δεξιά του. Ύστερα, καλεί τον εαυτό της για το πρώτο κομμάτι του πίνακα(μεχρι το piv) και το δεύτερο κομμάτι (μετα το piv). Λειτουργεί, δηλαδή, διαμερίζοντας ένα πίνακα σε δύο μέρη και ταξινομώντας αυτά τα μέρη, το ένα ανεξάρτητα από το άλλο. Φτάνουμε στην πλήρη ταξινόμηση κάνοντας πρώτα τη

διαμέριση και μετά εφαρμόζοντας αναδρομικά τη μέθοδο στους δυο υποπίνακες που προκύπτουν.

partition: Κατά τη διαμέριση, κάποιο στοιχείο του πίνακα array θεωρούμε ότι βρίσκεται στη τελική του θέση. Στο δικό μας αλγόριθμο το στοιχείο αυτό είναι πάντα το τελευταίο στοιχείο του πίνακα/υποπίνακα. Έτσι, στην αρχή θέτουμε ως pivot το array[end]. Έπειτα, με μια επανάληψη for ελέγχουμε απο το start μεχρι το end αν τα στοιχεία του πίνακα είναι μεγαλύτερα από το pivot. Αν η συνθήκη ισχύει, τοτε με τη βοήθεια της μεθόδου swap ανταλλάσουμε τα δυο στοιχεία. Η ιδέα είναι κανένα από τα στοιχεία πριν το pivot να μην είναι μεγαλύτερα από το pivot και αντίστοιχα κανένα στοιχείο μετά το pivot να μην είναι μικρότερο του. Τέλος, η συνάρτηση επιστρέφει τη θέση που έχει τοποθετηθεί το pivot.

Swap: Ανταλλάσσει τις θέσεις δύο στοιχείων στο πίνακα.

Μέρος Δ

Για την παραγωγή των δοκιμαστικών αρχείων φτιάξαμε την κλάση CreateFolders, η οποία παράγει 10 αρχεία, με όνομα Folders_N_i.txt, οπου N ο αριθμός των φακέλων που βρίσκονται σε κάθε αρχείο i = 1..10. Τα αρχεία που δημιουργούνται αποθηκεύονται στο φάκελο data, και στους subfolders που τους αντιστοιχούν για κάθε N.
Για την εγγραφή των φακέλων προς αποθήκευση σε δίσκους, δηλαδή την εγγραφή των MB κάθε φακέλου, χρησιμοποιήσαμε την κλάση Random. Αρχικά, κατασκευάσαμε το αντικείμενο random της κλάσης Random και μέσω της μεθόδου της nextInt(), παρήγαμε ακέραιους αριθμούς μεταξύ του 0 και του 1.000.000.
Παράγουμε έναν ακέραιο αριθμό και τον γράφουμε ταυτόχρονα στο αρχείο μας μέσω της μεθόδου write της κλάσης File.
Τη διαδικασία αυτή(η οποία περιγράφεται με κώδικα στη γραμμη 15) τη τρέχουμε μέσα σε μια επανάληψη for, για όσους φακέλους θέλει να δημιουργήσει ο χρήστης. Ο αριθμός των φακέλων δίνεται ως όρισμα απο τον χρήστη στη γραμμή εντολών, για να έχει την ελευθερία να δημιουργήσει και αρχεία με 10, 50, 600 φακέλους. Ωστόσο, τα αρχεία που παράγονται κάθε φορά για N φακέλους είναι σταθερά και μόνο 10 για τους σκοπούς της εργασίας, αν και εύκολα μπορεί να αλλάξει. Η δημιουργία των 10 φακέλων γίνεται σε μια επανάληψη for με i = 1..10, στην οποία βρίσκονται μέσα και όσα αναφέρθηκαν παραπάνω. Τέλος, αν τα αρχεία δημιουργηθούν με επιτυχία εμφανίζεται αντίστοιχο μήνυμα.

Η εκτέλεση του πειράματος, δηλαδή η σύγκριση των δύο αλγορίθμων, γίνεται στη κλάση Experiment.
Αρχικά, δημιουργούμε 3 αντικείμενα της κλάσης File, τα folders_100, folders_500 και folders_1000 που αναπαριστούν τους 3 φακέλους από τους οποίους θα διαβάσουμε τα δεδομένα. Ακόμα, για τη δική μας διευκόλυνση στη προσπέλαση των αρχείων φτιάξαμε τον πίνακα ακεραίων numbers, ο οποίος περιέχει τους αριθμους 100, 500 και 1000 των φακέλων, και τον πίνακα folders, ο οποίος περιέχει τα δεδομένα που διαβάζονται από κάθε αρχείο. Τέλος, για να μπορούμε να εμφανίσουμε συγκεντρωτικά τα αποτελέσματα των δυο αλγορίθμων έχουμε και τους πινακες Sum1, Sum2, στους οποίους αποθηκεύουμε το άθροισμα των δίσκων που χρησιμοποίησε κάθε αλγόριθμος για κάθε περίπτωση του N.

Σε μια επανάληψη for, για κάθε N, εκτελούμε την παρακάτω διαδικασία.

Αρχικοποιούμε τις μεταβλητες Sum1[i], Sum2[i] με 0, για κάθε i = 0,1,2. Ύστερα, ανάλογα με το N, αποθηκεύουμε στη μεταβλητή folder τύπου File ένα από τα folders_100, folders_500 και folders_1000, για να διαβάσουμε τα δεδομένα. Έπειτα, με τη βοήθεια της μεθόδου listFiles(), μπορούμε να προσπελάσουμε σε μια for τα αρχεία που βρίσκονται αποθηκευμένα στον εκάστοτε φάκελο.
Στον πίνακα ακεραίων folders αποθηκεύουμε τον πίνακα με τα δεδομένα που επιστρέφει η μέθοδος της κλάσης Greedy readFile, η οποία παίρνει σαν όρισμα το directory και το όνομα του αρχείου. Τέλος, τρέχουμε τον αλγόριθμο 1 με όρισμα τον πίνακα folders και προσθέτουμε στο Sum1 τον ακέραιο αριθμό που επιστρέφει. Ο αριθμός που επιστρέφει είναι ο αριθμός των δίσκων που χρειάστηκαν για την αποθήκευση των φακέλων. Τα ίδια κάνουμε και για τον αλγόριθμο 2. Λόγω του μέρους Δ, τόσο ο αλγόριθμος 1, όσο και ο αλγόριθμος 2 του Β μέρους δεν είναι void, όπως ζητείται, αλλά επιστρέφουν ένα ακέραιο αριθμό.
Τέλος, σε μια δεύτερη επανάληψη for, εκτυπώνονται συγκεντρωτικά οι μέσοι όροι δίσκων που χρειάστηκαν οι δύο αλγόριθμοι για κάθε N.

Από την εκτέλεση του πειράματος βγαίνουν τα παρακάτω αποτελέσματα:

```
---------------Comparison---------------
----------------For N = 100----------------

Average disks used for algorithm Greedy :           59.1

Average disks used for algorithm Greedy-Decreasing : 52.9
-----------------------------------------
----------------For N = 500----------------

Average disks used for algorithm Greedy :           293.1

Average disks used for algorithm Greedy-Decreasing : 256.6
-----------------------------------------
----------------For N = 1000----------------

Average disks used for algorithm Greedy :           587.3

Average disks used for algorithm Greedy-Decreasing : 509.3
```

Όπως φαίνεται, αν και είναι προφανές, ο αλγόριθμος Greedy-Decreasing χρησιμοποιεί πολύ λιγότερους φακέλους από ότι ο αλγόριθμος Greedy. Η ταξινόμηση απο τον μεγαλύτερο φάκελο στον μικρότερο, κάνει αισθητή την διαφορά μεταξύ των δύο αλγορίθμων, καθώς όσο αυξάνεται το N, τόσο μεγαλύτερη είναι η διαφορά των δίσκων που χρησιμοποιήθηκαν. Τέλος, από άποψη απόδοσης, ο αλγόριθμος Greedy-Decreasing, φαίνεται πιο αποδοτικός, καθώς δεν δεσμεύει άσκοπα μνήμη, δημιουργώντας νέους δίσκους για την αποθήκευση φακέλων, ενώ χωράνε σε κάποιο προηγούμενο. Ωστόσο για πολύ μεγαλύτερα N, θα πρέπει να ληφθεί υπόψη τόσο ο χρόνος, όσο και οι πόροι που χρειάζονται για να γίνει η ταξινόμηση.

Τέλος, παραθέτουμε και το διάγραμμα ανάλυσης των δύο αλγορίθμων.