3η ΕΡΓΑΣΙΑ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΝΙΚΟΛΕΤΑ-ΚΛΕΙΩ ΠΑΤΑΤΣΗ 3180266
ΠΛΑΤΩΝ ΚΑΡΑΓΕΩΡΓΗΣ 3180068

ΑΝΑΛΥΣΗ ΜΕΘΟΔΩΝ

- void insert(Suspect item): Εδώ ακολουθούμε τις εντολές της εκφώνησης και χρησιμοποιούμε την insert που προσθέτει το νέο στοιχείο σαν ρίζα με χρήση της random. Πρώτα γίνεται ένας έλεγχος μέσω της searchR για το αν υπάρχει ήδη το στοιχείο που μας ζητείται να προσθέσουμε στο δέντρο. Αν υπάρχει ήδη τυπώνεται μήνυμα λάθους. Σε διαφορετική περίπτωση καλείται η βοηθητική συνάρτηση insertAsRoot() η οποία φροντίζει να γίνουν οι περιστροφές που χρειάζονται για να φτιαχτεί σωστά το δέντρο καλώντας την insert() που με τη σειρά της καλεί τις rotR, rotL. Θεωρήσαμε ότι έπρεπε να γίνεται print που να ενημερώνει την εξέλιξη όταν ο user κάνει insert, γι'αυτό και φτιάχνουμε 2 μεταβλητές helping_hand και helping_hand_2 που φροντίζουν να γίνονται τα prints μόνο όταν το insert το ζητά ο χρήστης και όχι σε κάθε insert όπως πχ τα αρχικά.

  Πολυπλοκότητα: **O(N)** γιατί είμαστε σε δυαδικό δέντρο και αυτή είναι η μέση πολυπλοκότητα. Μάλιστα, η κατασκευή που κάνουμε αρχικά καλώντας Ν φορές την insert όπου Ν το μέγεθος του αρχείου, έχει πολυπλοκότητα **O(NlogN)** κατά μέσο όρο.

- void load(String filename): Διαβάζουμε το αρχείο που θα δώσει ως input ο χρήστης. Για να αποφύγουμε διπλό διάβασμα του δίνουμε το maximum length στο for loop (μέχρι 99999 ΑΦΜ που είναι και μοναδικά) και του λέμε ότι με το που διαβάσει null String να κάνει break. Επίσης, για να μην επέμβουμε στο έτοιμο interface του δίνουμε με μία εντολή που βρήκαμε το exception με το catch, για να γλιτώσουμε το «throws IOException» που έπρεπε να προστεθεί στο interface.

  Πολυπλοκότητα: **O(N)** γιατί έχουμε ένα for loop το οποίο θα διαβάσει τις Ν εισόδους και μόλις τελειώσουν θα κάνει break.

- void updateSavings(int AFM): Καλείται η searchR() αν υπάρχει ο ύποπτος του ανανεώνει τα savings ανάλογα με το input του χρήστη, αν όχι τυπώνει ανάλογο μήνυμα λάθους και δεν πειράζει τίποτα.

  Πολυπλοκότητα: **O(logN)** γιατί καλεί searchR() που είναι η μέθοδος αναζήτησης σε ΔΔΑ. Έχει ωστόσο χειρότερη περίπτωση **O(N)**.

- Suspect searchByAFM(int AFM): Καλεί την searchR() που θα αναλυθεί πιο κάτω και αυξάνει τις δύο βοηθητικές μεταβλητές για να είναι σωστά τα prints αργότερα στις άλλες μεθόδους που χρησιμοποιούν την searchR().

  Πολυπλοκότητα: **O(logN)** γιατί καλεί searchR() που είναι η μέθοδος αναζήτησης σε ΔΔΑ. Έχει ωστόσο χειρότερη περίπτωση **O(N)**.

- private Suspect searchR(TreeNode r, int v): Η μέθοδος αυτή δεν υπάρχει στη λίστα που μας δίνεται, είναι βοηθητική, αλλά η συνεισφορά της είναι μεγάλη σε πολλές μεθόδους. Ουσιαστικά είναι μία μέθοδος διάσχισης που ψάχνει όλο το δέντρο για να εντοπίσει έναν εγκληματία με το ΑΦΜ που της δίνεται ως όρισμα. Η λειτουργία της είναι όπως στις διαφάνειες με μία προσθήκη, υπάρχουν 2 else if για να διαχωρίζονται τα εξής: Πότε κλήθηκε να ψάξει κάτι από το πρόγραμμα και πότε κλήθηκε να ψάξει κάτι από το χρήστη. Όταν κλήθηκε από το πρόγραμμα, σημαίνει ότι δεν θέλουμε ανάλογα prints και όταν κλήθηκε από το χρήστη (έμμεσα) σημαίνει ότι θέλουμε να γίνουν κάποια prints και γι'αυτό βάζουμε επιπλέον ελέγχους που θα τηρούνται μόνο όταν έχουν κληθεί από το χρήστη, γιατί επηρεάζουμε τις μεταβλητές από τη Main.

  Πολυπλοκότητα: **O(logN)** γιατί είναι η μέθοδος αναζήτησης σε ΔΔΑ. Έχει ωστόσο χειρότερη περίπτωση **O(N)**.

- List searchByLastName(String last_name): Φτιάχνουμε μία κλάση List που είναι μία απλή λίστα μονής σύνδεσης και την αρχικοποιούμε πάνω στη RandomizedBST. Καλεί τη searchEveryNode() που είναι ίδιας λειτουργίας με την searchR απλά εδώ ψάχνει με βάση το String που έδωσε σαν input ο χρήστης. Κάθε επώνυμο που ταιριάζει με ένα του δέντρου το προσθέτει στη λίστα. Αφού γίνει το search, αν η λίστα έχει μέγεθος 0<n<=5 τότε κάνει print τα στοιχεία όλων αυτών που έχουν αυτό το επώνυμο. Αν δεν βρει κανέναν με αυτό το επώνυμο κάνει print ότι η λίστα είναι null. Αν είναι παραπάνω από 5 κάνει print ακριβώς αυτό, χωρίς να κάνει print τα στοιχεία τους.

  Πολυπλοκότητα: **O(N)** γιατί καλεί τη searchEveryNode() που είναι μέθοδος διάσχισης και ψάχνει κάθε κόμβο ανάλογα με το String, επομένως εδώ δεν έχουμε O(logN) διότι δεν μπορούμε να χρησιμοποιήσουμε την ιδιότητα του ΔΔΑ, αφού τα στοιχεία είναι κατανεμημένα με βάση το ΑΦΜ και όχι το επώνυμο.

- public void remove(int AFM): Ψάχνει να βρει αρχικά με τη searchR αν υπάρχει το ΑΦΜ που της δόθηκε σαν όρισμα μέσα στη λίστα. Αν δεν το βρει τυπώνει ανάλογο μήνυμα. Αν το βρει κάνει την αφαίρεση, ενημερώνει τις μεταβλητές και τυπώνει ανάλογο μήνυμα για την επιτυχία της διαγραφής. Μετά, καλεί τη removeR που με τη σειρά της καλεί τη joinLR. Η joinLR δεν έχει επεξεργαστεί και είναι ακριβώς όπως την όρισαν οι διαφάνειες.

Πολυπλοκότητα: **O(logN)** κατά μέση περίπτωση αλλά μπορεί να είναι και **O(N)**. Γενικά εξαρτάται από το ύψος του δέντρου που κυμαίνεται μεταξύ αυτών των δύο τιμών.

- double getMeanSavings(): Φτιάχνουμε ένα βοηθητικό array το οποίο αρχικοποιούμε μέσα σε αυτή τη μέθοδο και του δίνουμε για size το άθροισμα των στοιχείων που είναι μέσα στο δέντρο. Έπειτα καλούμε την crossTree() στην οποία κάνουμε άλλη μία διάσχιση και συμπληρώνουμε το array που φτιάξαμε. Στο τέλος το array έχει όλα τα savings και σε ένα for loop παρακάτω υπολογίζουμε τη μέση τιμή.

  Πολυπλοκότητα: **O(N)** γιατί καλεί την crossTree που διασχίζει κάθε κόμβο για να πάρει τα savings του.

- void printTopSuspects(int k): Εδώ έχουμε κάνει το εξής. Πρώτα βάζουμε όλους τους υπόπτους σε ένα array. Μετά πάμε και παίρνουμε αυτούσιο το Heapsort της $2^{ης}$ εργασίας και το προσαρμόζουμε για να λειτουργήσει σε αυτή την περίπτωση. Πλέον το μόνο που χρειαζόμαστε είναι ένα Comparator. Αυτό το τοποθετούμε στη κλάση Suspect και μάλιστα του δίνουμε 2 διαφορετικές λειτουργίες ανάλογα με ένα «σημαφόρο». Επειδή θα χρειαστούμε το sort και για την printByAFM του λέμε ότι εάν ο σημαφόρος είναι 1 σημαίνει ότι σε κάλεσε η printTopSuspects και τότε θέλουμε να χρησιμοποιήσεις το comparison με το δηλωμένο εισόδημα<9000 και με τη διαφορά δηλωμένου εισοδήματος και πραγματικού εισοδήματος. Για αυτά μάλιστα φτιάχνουμε κάποιους getters για ευκολία. Όταν γίνει το sort τότε απλά κάνουμε τα ανάλογα prints με ένα for loop.

  Πολυπλοκότητα: Η διαδικασία που γίνεται με την crossTree για να γεμίσει το array που δίνουμε στη Heapsort είναι O(N). Η Heapsort θέλει O(nlogn). Το for loop για τα print θέλει O(N) γιατί το k αν είναι μεγαλύτερο το κάνω ίσο με N. Άρα O(N + nlogn + N) = **O(NlogN)** πολυπλοκότητα.

- void printByAFM(): Αρχικά θέλουμε να συλλέξουμε όλα τα ΑΦΜ και να τα βάλουμε σε έναν πίνακα. Για εξοικονόμηση κώδικα χρησιμοποιούμε τον ίδιο πίνακα που χρησιμοποιούμε παραπάνω στην printTopSuspects. Για εξοικονόμηση κώδικα επίσης χρησιμοποιούμε και την ίδια μέθοδο που χρησιμοποιήσαμε για να συλλέξουμε και τα savings, την μέθοδο διάσχισης crossTree. Έτσι εξηγείται και η μεταβλητή caller που έχουμε, καθώς όταν είναι 1 τότε μπαίνει στο ένα if που συλλέγει τα ΑΦΜ και όταν είναι 0, δηλαδή την crossTree την κάλεσε η getMeanSavings, συλλέγει τα savings. Τέλος, όπως εξηγήσαμε και παραπάνω στην printTopSuspects για να γίνει το sort χρησιμοποιούμε ένα σημαφόρο, ο οποίος τώρα έχει την τιμή 0 άρα όταν καλέσουμε τη Heapsort και κατ'επέκταση το comparator θα γίνει η σύγκριση με βάση τα ΑΦΜ. Ακολουθεί ένα for loop που κάνει τα ανάλογα print.

  Πολυπλοκότητα: Η διαδικασία που γίνεται με την crossTree για να γεμίσει το array που δίνουμε στη Heapsort είναι O(N). Η Heapsort θέλει O(nlogn). Το for

loop για τα print θέλει O(N) επίσης, άρα O(N + nlogn + N) = **O(NlogN)** πολυπλοκότητα.