Δομές Δεδομένων: Εργασία 3

Μέθοδοι του RandomizedBST

ΑΠΌ ΒΑΣΙΛΗΣ ΠΑΠΑΔΗΜΑΣ (3220150) & ΜΑΡΙΟΣ ΜΑΤΣΑ (3220120)

void insert(LargeDepositor item)

Χρησιμοποιούμε την TreeNode foundByAFM(int AFM) για να βρούμε αν υπάρχει ήδη κάποιο TreeNode στο δέντρο με ίδιο ΑΦΜ με του item (O(N)). Αν ναι (curr != null) τότε εκτυπώνουμε το μήνυμα, αλλιώς θέτουμε root = insertAsRoot(root, item) και αυξάνουμε το N του root. Η TreeNode insertAsRoot(TreeNode h, LargeDepositor x) (O(log_2(N))) και η TreeNode insertT(TreeNode h, LargeDepositor x) (O(log_2(N))) που χρησιμοποιεί (και οι TreeNode rotR(TreeNode h) (O(1)) και TreeNode rotL(TreeNode h) (O(1)) που χρησιμοποιεί εκείνη) έχουν υλοποιηθεί όλες σύμφωνα με τα παραδείγματα που έχουν δοθεί στα πλάισια του μαθήματος.

Time complexity: O(log_2(N))

void load(String filename)

Διαβάζουμε σειριακά το αρχείο που ορίζει το filename (O(N)), σπάμε την κάθε γραμμή σε substrings με spacer τα κενά μέσω regex (O(μήκος γραμμής για regex που έχει μετατραπεί σε finite automaton με υπόθεση 3 spacers ανα γραμμή), πρακτικά σταθερός όρος O(1)), δημιουργούμε αντικείμενα LargeDepositor με τα κατάλληλα δεδομένα από τα strings (που όταν χρειάζεται μετατρέπονται σε άλλες δομές δεδομένων) (O(1)) και καλούμε την insert με αυτά ως όρισμα (O(log_2(N)), το πραγματικό N αλλάζει σε κάθε επανάληψη αλλά upper bound to N). Αν προκύψει κάποιο Exception σε αυτή την διαδικασία, κάνουμε throw RuntimeException.

Time complexity: O(N*log_2(N))

List searchByLastName(String last_name)

Δημιουργούμε μια List για να αποθηκεύσουμε το αποτέλεσμα της αναζήτησης O(1). Η υλοποιήση της List είναι βασισμένη στον σχετικό κώδικα που γράψαμε για μια προηγούμενη εργασία του μαθήματος, με ελαφρές αλλαγές. Η αναζήτηση γίνεται με αναδρομικές κλήσεις της void traverseAndBuild(TreeNode node, List ls), η οποία κάνει inorder traversal του BST (O(N)) και προσθέτει τον κόμβο στον οποίο βρίσκεται στην List (O(log_2(N))) αν έχει ίδιο επώνυμο με το όρισμα last_name (το οποίο για ευκολία έχει αποθηκευτεί σε ένα ειδικό πεδίο της List). Όταν τελειώσει η traverseAndBuild, ελέγχουμε πόσα στοιχεία έχει η ls (με το πεδίο της N, O(1)) και αν είναι μεταξύ του 0 και του 5 τα εκτυπώνουμε O(N_λίστας) πριν την επιστρέψουμε.

Time complexity: O(N*log_2(N))

double getMeanSavings()

Επιστρέφουμε το αποτέλεσμα της κλήσης της double traverseAndSum(TreeNode node), η οποία με αναδρομικές κλήσεις κάνει inorder traversal (O(N)) του BST (φυσικά δεν έχει σημασία η σειρά διασχίσης) και αθροίζει και επιστρέφει τα savings όλων των LargeDepositor που υπάρχουν στο δέντρο, διαιρεμένο με το πεδίο N του BST (O(1)).

Time complexity: O(N)

void printTopLargeDepositors(int k)

Δημιουργούμε ενα priority queue **pq** με capacity k (η υλοποίηση της κλάσης PQ είναι από προηγούμενη εργασία, με την διαφορά ότι κάναμε το capacity σταθερό για διευκολύνση στην χρήση της σε αυτό το πλαίσιο) και μια List που θα χρειαστούμε στην εκτύπωση των αποτελεσμάτων.

Καλούμε την void traverseAndRank(TreeNode node, PQ pq), η οποία αναδρομικά διασχίζει το δέντρο (O(N), πάλι in-order χωρίς να κάνει διαφορά η σειρά) και προσθέτει (O(log_2(N))) το LargeDepositor που είναι αποθηκευμένο στο TreeNode που βρίσκεται σε κάθε κλήση της, εφόσον η pq δεν έχει γεμίσει, ή διαφορετικά εάν το score του τρέχοντος LargeDepositor είναι υψηλότερο από το score του min της pq (ο comparator της τοποθετεί σε αυτή την θέση το LargeDepositor με το μικρότερο score από αυτά της pq), στην οποία περίπτωση πρώτα αφαιρεί το ελάχιστο στοιχείο από την pq (O(log_2(N))) ώστε να αποφευχθεί το overflow της.

Μολίς τελειώσει η traverseAndRank, με ένα loop O(N) αποθηκεύουμε τα LargeDepositor του pq στην List, χρησιμοποιόντας την ουσιαστικά σαν ένα stack, αφού κάνουμε insertAtFront τα στοιχεία με αύξουσα σειρά κατά score, και μέτα τα κάνουμε removeFromFront με φθίνουσα σειρά, παράλληλα εκτυπώνοντας τα.

Time complexity: O(N*log_2(N))

void updateSavings(int AFM, double savings)

Αναζητούμε τον LargeDepositor με το δοσμένο ΑΦΜ μέσα από την μέθοδο foundByAFM(int AFM), η οποία εκτελεί δυαδική αναζήτηση με κλειδί το ΑΦΜ. Αν τον βρούμε ενημερώνουμε το πεδίο savings αλλιώς εκτυπώνουμε ανάλογο μήνυμα.

Σε κάθε περίπτωση λόγω της δυαδικής αναζήτησης η μέθοδος έχει **Time Complexity** -> O(logn), όπου n ο αριθμός κόμβων του δέντρου.

LargeDepositor searchByAFM(int AFM)

Αναζητούμε τον LargeDepositor με το δοσμένο ΑΦΜ μέσα από την μέθοδο foundByAFM(int AFM), η οποία εκτελεί δυαδική αναζήτηση με κλειδί το ΑΦΜ. Αν τον βρούμε τον επιστρέφουμε αλλιώς εκτυπώνουμε ανάλογο μήνυμα και επιστρέφουμε null.

Σε κάθε περίπτωση λόγω της δυαδικής αναζήτησης η μέθοδος έχει **Time Complexity**-> O(logn), όπου n ο αριθμός κόμβων του δέντρου

void remove(int AFM)

Αναζητούμε τον LargeDepositor με το δοσμένο ΑΦΜ μέσα από την μέθοδο foundByAFM(int AFM). Αν δεν τον βρούμε, δεν γίνεται τίποτα. Αν όμως τον βρούμε, τότε θα πρέπει να πάρουμε μια τυχαία απόφαση για το ποιο από τα παιδιά του θα μπει στην θέση του. Για την διαδικασία της ένωσης χρησιμοποιούμε την μέθοδο TreeNode joinNode(TreeNode a, TreeNode b), ενώ η ταχύτητα της μεθόδου εξαρτάται από το ύψος του δέντρου και άρα αφού έχουμε Τυχαιοποιημένο ΔΔΑ (άρα ισοζυγισμένο δέντρο) το ύψος θα είναι περίπου logn.

Time Complexity -> O(logn)