ΕΡΓΑΣΙΑ ΔΟΜΩΝ ΔΕΔΟΜΕΝΩΝ 2021-2022

Κωνσταντίνος Μπαζιόπουλος 3635

Μιχάλης Καραούλας 3747

Εισαγωγή

Στην παρακάτω έκθεση θα αναλυθεί η σχεδιαστική λογική του προγράμματος και θα παρουσιαστούν τα προβλήματα που αντιμετωπίστηκαν κατά την υλοποίησή του, καθώς και οι λύσεις που βρέθηκαν για αυτά.

Η έκθεση χωρίζεται σε 6 μέρη, ένα για κάθε μία από τις δομές, και ένα για τις «εξωτερικές λειτουργείες» του προγράμματος (είσοδος-έξοδος, χρονομέτρηση, κλπ.). Σε κάθε μέρος θα δίνονται και τα αντίστοιχα αρχεία κώδικα.

1.Εξωτερικές Λειτουργείες

Σε αυτό το μέρος αντιμετωπίζονται τα εξής θέματα.

- Ανάγνωση του αρχείου κειμένου
- Διαμόρφωση του αρχείου κειμένου (αφαίρεση στίξης και κεφαλαίων γραμμάτων)
- Καταμέτρηση Χρόνου

- Επιλογή του συνόλου Q
- Καταγραφή αποτελεσμάτων σε αρχείο εξόδου

Αναφερόμαστε σε κώδικα που υλοποιείται στα αρχεία: main.cpp και FileStuff.h

Από εδώ και στο εξής, ακολουθώντας τη λογική του κώδικα, θα αναφερόμαστε στο αρχείο εισόδου ως **inText** και στο αρχείο εξόδου ως **outText.** Για να αλλάξει κάποιος το αρχείο εισόδου (ή το όνομα του αρχείου εξόδου) αρκεί να βρει το #define inText στην αρχή του main.cpp.

Τα αρχεία στο πρόγραμμα χειρίζονται ως «ροές» με τη βιβλιοθήκη <fstream>. Σε πρώτη φάση θέλουμε να διαμορφώσουμε το αρχείο με βάση τις προδιαγραφές της εκφώνησης. Αυτό επιτυγχάνεται με την συνάρτηση formatFile(string filename), στην οποία δίνουμε ως όρισμα το inText. Αυτή δημιουργεί ένα νέο αρχείο με όνομα “formattedText.txt” (στο εξής **ft**), στο οποίο «αδειάζουμε» τα περιεχόμενα του inText χαρακτήρα-προς-χαρακτήρα, μετατρέποντας τα κεφαλαία σε πεζά, και παραλείποντας σημεία στίξης. Εδώ πρέπει να αναφέρουμε και την *Μεγαλύτερη Δυσκολία* της σχεδιαστικής διαδικασίας. Η formatFile() είναι σχεδιασμένη με βάση τον τύπο char και κατ’ επέκταση το σύστημα ASCII. Αυτό σημαίνει πως αδυνατεί να χειριστεί την κωδικοποίηση UTF-8 που χρησιμοποιείται για το αρχείο gutenberg.txt, ενώ είναι επαρκής για το small-file.txt. Παρά τις προσπάθειες από μέρους μας να την προσαρμόσουμε για τις ανάγκες του κειμένου, το θέμα αποδείχθηκε να ξεπερνά το γνωστικό μας επίπεδο. Αυτό σημαίνει πως αυτό το κομμάτι του κώδικα μπορεί να δώσει λάθος δεδομένα, **όμως οι λειτουργείες του προγράμματος δουλεύουν σωστά πάνω σε αυτά τα λάθος δεδομένα**, οπότε αποφασίσαμε να το αφήσουμε ως έχει, εφόσον ο κύριος σκοπός της εργασίας δεν είναι η επεξεργασία κειμένου, αλλά η υλοποίηση των δομών.

Με το ft έτοιμο μπορούμε να προχωρήσουμε στην καταχώρηση δεδομένων. Με μία βοηθητική συνάρτηση παίρνουμε τον αριθμό λέξεων στο κείμενο, που χρειάζεται για τις στατικές δομές μας. Έπειτα χρησιμοποιούμε την συνάρτηση

loadInto(tclass structure, string filename, int wc),(με filename ft) η οποία διαβάζει ανά δύο τις λέξεις του ft και τις εισάγει στη δομή structure. Η υλοποίησή της έγινε με πρότυπο, έτσι ώστε να μπορεί να χρησιμοποιηθεί με όλες τις κλάσεις και η μόνη προϋπόθεση είναι να υλοποιείται η συνάρτηση insert(string s) για κάθε δομή. Τέλος το wc είναι ο αριθμός λέξεων και χρησιμοποιείται μόνο για να ενημερώνει το χρήστη για την πρόοδο της διαδικασίας καταχώρησης, μέσω της γραμμής εντολών (δηλαδή δεν αφορά την εργασία και βρίσκεται εκεί μόνο για λόγους debugging)

Η καταμέτρηση χρόνου γίνεται με την βιβλιοθήκη <chrono> όπως περιγράφεται σε αυτό το άρθρο: https://www.geeksforgeeks.org/measure-execution-time-function-cpp/. Η μόνη τροποποίηση που έγινε (για λόγους αισθητικής) ήταν ο ορισμός *μακροεντολών* **STARTTIMER** και **STOPTIMER** έτσι ώστε να φαίνεται ξεκάθαρα που ξεκινάει και πού τελειώνει η καταμέτρηση. Θα εξηγούσαμε περεταίρω τη λειτουργεία αυτή, αλλά το μόνο που ξέρουμε είναι ότι δουλεύει έτσι όπως είναι και δεν χρειάζεται να αλλάξει…

Το σύνολο Q επιλέγεται τυχαία με τη συνάρτηση string* getQ(UnorderedArray &ua), η οποία παράγει 1000 τυχαίες ακέραιες τιμές και αποθηκεύει τα ζεύγη σε με αυτούς τους δείκτες από τον μη-ταξινομημένο πίνακα *πριν αυτός διαμορφωθεί* (περισσότερες λεπτομέρειες στο μέρος 2)

Συνοψίζοντας, έχουμε μεθόδους για να διαμορφώνουμε το αρχείο εισόδου, να εξάγουμε τα ζεύγη από το ft, να μετράμε χρόνο σε οποιαδήποτε διαστήματα της εκτέλεσης χρειάζεται και να διαλέγουμε ένα σύνολο από 1000 ζεύγη με τυχαίο τρόπο.

Προχωράμε στον κύριο βρόγχο του προγράμματος, ο οποίος είναι ο εξής:

Για κάθε δομή

- γράφουμε το όνομά της στο outText με κεφαλαία και δημιουργούμε ένα αντικείμενό της (εάν η δομή είναι στατική μετράμε το χρόνο που χρειάστηκε για τη δέσμευση μνήμης και τον βάζουμε στο outText)

- εισάγουμε όλα τα ζεύγη λέξεων από το inText καλώντας την loadInto() και γράφουμε τον χρόνο καταχώρησης στο outText
- εκτελούμε 1000 αναζητήσεις με τα περιεχόμενα του Q και γράφουμε τα αποτελέσματά τους στο outText και στο τέλος το συνολικό χρόνο αναζητήσεων (αυτός ο χρόνος συμπεριλαμβάνει και τις καταγραφές στο outText αλλά μπορούμε να τις θεωρήσουμε αμελητέες)
- διαγράφουμε τη δομή πριν συνεχίσουμε με την επόμενη (καθώς για πολύ μεγάλα αρχεία κειμένου, θα ξεμέναμε από μνήμη πριν προλάβουμε να αποθηκεύσουμε όλες τις δομές ταυτόχρονα)

Η μόνη δομή που χειρίζεται λίγο διαφορετικά είναι ο αταξινόμητος πίνακας, ο οποίος αξιοποιείται και για το σύνολο Q, αλλά χρειάζεται και ένα ακόμα βήμα που περιγράφεται στο μέρος 2.

Με δεδομένα τα παραπάνω, οι δομές που υλοποιούμε χρειάζονται τουλάχιστον συναρτήσεις insert() και search(), καταστροφέα και συναρτήσεις πρόσβασης στα ζεύγη και στον αριθμό εμφανίσεων για κάθε τους δεδομένο(getString() και getAmount() στις οποίες δεν θα αναφερθούμε γιατί δεν έχουν κάποιο ενδιαφέρον). Επίσης τα ζεύγη σε κάθε δομή χειρίζονται ως απλά strings καθώς αυτή η μέθοδος δουλεύει ικανοποιητικά.

2.Αταξινόμητος Πίνακας

Αναφερόμαστε σε κώδικα από τα αρχεία: UnorderedArray.h και UnorderedArray.cpp

Ορίζουμε κλάση UnorderedArray που αποτελείται από δύο pointers (string* data και int* amount), και δυο int, size(συνολικές θέσεις) και next(η επόμενη διαθέσιμη θέση)

Κατασκευή

Όταν δημιουργούμε αντικείμενα αυτής της κλάσης δίνουμε πρώτα το μέγεθος που χρειαζόμαστε (στην προκειμένη περίπτωση τον αριθμό λέξεων - 1) και σύμφωνα με αυτό δεσμεύονται δύο πίνακες, ένας για strings και ένας για int, οι αρχές των οποίων ανατίθενται στους δείκτες data και amount αντίστοιχα. Το next παίρνει την τιμή 0 αφού η επόμενη εισαγωγή θα είναι στην πρώτη θέση των πινάκων

Εισαγωγές

Οι εισαγωγές στον αταξινόμητο πίνακα ήταν το μόνο σημείο στο οποίο χρειάστηκε να αυτοσχεδιάσουμε. Εφόσον χρειάζεται να αποθηκεύουμε το κάθε ζεύγος μία φορά και να ανανεώνουμε τις εμφανίσεις του κάθε φορά που το ξαναβρίσκουμε, η εμφανής λύση θα ήταν να κάναμε μία αναζήτηση για κάθε εισαγωγή και να καταχωρούσαμε νέο ζεύγος μόνο αν η αναζήτηση αυτή αποτύγχανε. Αυτή η μέθοδος όμως έχει δύο προβλήματα:

1. Η είσοδος σε αταξινόμητο πίνακα έχει κανονικά πολυπλοκότητα O(1), η οποία θα γινόταν O(n) μαζί με τις αναζητήσεις
2. Η εκτέλεση τόσων πολλών σειριακών αναζητήσεων ακούγεται παρανοϊκή…

Η λύση: οι εισαγωγές γίνονται «τυφλά», με κάθε ζεύγος να αποθηκεύεται αρχικά, όσες φορές εμφανίζεται (σε κάθε καταχώρηση ο αριθμός εμφανίσεων τίθεται ως 1). Αφού εισαχθούν όλα τα δεδομένα τρέχουμε μία άλλη συνάρτηση που ονομάζουμε **συμπίεση** (compress()). Κατά την συμπίεση επιλέγουμε ένα αντικείμενο (ξεκινώντας από το πρώτο και συνεχίζοντας με τη σειρά) και (ξεκινώντας από την επόμενη θέση) διατρέχουμε τον πίνακα μέχρι το τέλος. Όποτε βρίσκουμε αντίγραφα τα «διαγράφουμε» (θέτουμε τον αριθμό επαναλήψεών τους σε -1, έτσι ώστε να μην τα ξαναελέγχουμε σε επόμενες διαπεράσεις).

Η σύγκριση των δύο μεθόδων (θα τις ονομάσουμε **προ-αναζήτηση** και **συμπίεση** για λόγους κατανόησης) με μαθηματικό τρόπο είναι δύσκολη, καθώς έχουν την

ίδια χειρότερη περίπτωση (όλα τα δεδομένα είναι διαφορετικά) και σε αυτή τη χειρότερη περίπτωση έχουν ίδιο κόστος (μία *άσκοπη* σειριακή αναζήτηση για κάθε στοιχείο, η οποία ελέγχει {1,2,..,n} θέσεις για την προ-αναζήτηση και {n,n-1,…1} θέσεις για την συμπίεση). Όμως νοηματικά, η μέθοδος της συμπίεσης εκτελεί μία σειριακή διαπέραση του πίνακα για κάθε **μοναδικό** στοιχείο και μάλιστα, όσο πιο πολλές εμφανίσεις έχει το στοιχείο, τόσο μειώνεται ο χώρος αναζήτησης. Η μέθοδος της προ-αναζήτησης από την άλλη χρειάζεται μία σειριακή αναζήτηση για κάθε στοιχείο που εισάγεται. Οπότε διαλέξαμε την μέθοδο της συμπίεσης.

Μία λεπτομέρεια εδώ είναι πως όσο πιο νωρίς μετρήσουμε τα ζεύγη με μεγάλο αριθμό επαναλήψεων, τόσο πιο αποδοτικά θα εκτελούνται οι επόμενες αναζητήσεις. Σε κάποια φάση της σχεδίασης βρήκαμε τα πιο συνηθισμένα ζεύγη λέξεων στην Αγγλική γλώσσα και να συμπιέσαμε με βάση αυτά πρώτα, όμως στο τέλος αποφασίσαμε να μην υλοποιήσουμε το πρόγραμμα έτσι.

Τέλος, πριν γίνει η συμπίεση, ο πίνακας έχει αποθηκευμένο ένα τέλειο αντίγραφο του κειμένου, ανά ζευγάρια λέξεων, οπότε πριν γίνει η συμπίεση, τον αξιοποιούμε για την παραγωγή του συνόλου Q.

Αναζήτηση

Η αναζήτηση στον αταξινόμητο πίνακα γίνεται μόνο σειριακά. Η μόνη σημείωση που πρέπει να κάνουμε είναι πως παραλείπουμε τις συγκρίσεις με στοιχεία που έχουν αριθμό εμφανίσεων -1, καθώς γνωρίζουμε πως δεν είναι έγκυρα.

Καταστροφή

Για να διαγράψουμε αυτή τη δομή αρκεί να αποδεσμεύσουμε τους δύο πίνακες μας.

3.Ταξινομημένος Πίνακας

Αναφερόμαστε σε κώδικα από τα αρχεία: OrderedArray.h και OrderedArray.cpp

Ορίζουμε κλάση OrderedArray που περιέχει δύο pointers (string* data και int* amount) και δύο int (size, συνολικός αριθμός θέσεων, last, τελευταία θέση που χρησιμοποιήθηκε)

Κατασκευή

Ο κατασκευαστής δέχεται ως όρισμα το μέγεθος και δεσμεύει την απαραίτητη μνήμη για τα data και amount. Επίσης θέτει το last ως -1.

Μετακίνηση

Υλοποιούμε την συνάρτηση shift(int start) που μετακινεί όλα τα δεδομένα του πίνακα (των δύο πινάκων μάλλον) μία θέση στα δεξιά ξεκινώντας από το δείκτη start. Αυτή η συνάρτηση μας χρειάζεται για τις εισαγωγές.

Αναζήτηση

Αξιοποιούμε τον αλγόριθμο της δυαδικής αναζήτησης για να βρούμε τη θέση αντικειμένων και αν αυτά ανήκουν στη δομή.

Η ανάγκη για αναζήτηση πριν κάθε εισαγωγή μας οδηγεί σε μία τροποποίηση. Η συνάρτησή μας δέχεται με αναφορά έναν ακέραιο pos ο οποίος, όταν η αναζήτηση αποτυγχάνει, παίρνει ως τιμή τη θέση στην οποία πρέπει να εισάγουμε το δεδομένο που αναζητήσαμε, αν θελήσουμε να το εισάγουμε. Έτσι κατά τις εισαγωγές καλούμε μόνο μία αναζήτηση και γνωρίζουμε την κατάλληλη θέση. Η συνάρτηση αναζήτησης όμως είναι υπερφορτωμένη, έτσι ώστε να μπορεί ο προγραμματιστής να αγνοήσει το pos όταν εκτελεί αναζητήσεις χωρίς να θέλει να κάνει και εισαγωγές.

Η ίδια ιδέα αξιοποιείται και στις άλλες δομές (εκτός του αταξινόμητου πίνακα και του πίνακα κατακερματισμού), αλλά στο εξής δεν θα την αναφέρουμε.

Εισαγωγή

Πρώτα κάνουμε αναζήτηση με την παραπάνω συνάρτηση. Αν η αναζήτηση είναι επιτυχής, τότε στη θέση που επιστρέφει, αυξάνουμε το amount. Διαφορετικά παίρνουμε την κατάλληλη θέση από το pos, κάνουμε shift(pos) και εισάγουμε το νέο string με amount 1.

Καταστροφή

Αποδεσμεύουμε τα amount και data.

4.Δυαδικό Δέντρο Αναζήτησης

Αναφερόμαστε σε κώδικα από τα αρχεία: BST.h και BST.cpp

Ορίζουμε κλάση BST με BST* left, BST* right, string data, int amount. Τα αντικείμενα αυτής της κλάσης αντιπροσωπεύουν κόμβους ενός δυαδικού δέντρου, αλλά σκοπός μας ήταν, ο προγραμματιστής να χρησιμοποιεί μόνο ένα αντικείμενο για να αναφέρεται στο σύνολο. Εξηγούμε παρακάτω πως επιτυγχάνεται αυτό.

Κατασκευή

Έχουμε δύο κατασκευαστές. Ο ένας χρησιμοποιείται εκτός της κλάσης και δεν παίρνει ορίσματα. Αυτός δημιουργεί τον ριζικό κόμβο του δέντρου με amount 0 και data “root”(η συγκεκριμένη τιμή δεν έχει σημασία). Ο κόμβος αυτός δεν λαμβάνεται υπ’ όψη στις υπόλοιπες λειτουργείες και δρα ως «διεπαφή» ανάμεσα στον προγραμματιστή και το σύνολο.

Ο άλλος κατασκευαστής παίρνει όρισμα string και εκτελείται μόνο όταν είναι απαραίτητο κατά την εισαγωγή. Θέτει το data ανάλογα με το όρισμά του και amount 1.

Και στους δύο κατασκευαστές τα pointers left και right αρχικοποιούνται ως null και αλλάζουν κατά την εισαγωγή.

Αναζήτηση

Ο αλγόριθμος αναζήτησης για δυαδικά δέντρα είναι καλά ορισμένος στη θεωρία. Ξεκινώντας από τη ρίζα αναζητούμε αναδρομικά στα παιδιά του κάθε κόμβου μέχρι να βρεθεί το κλειδί ή κάποιο φύλλο του δέντρου.

Εισαγωγή

Εκτελούμε αναζήτηση. Σε περίπτωση επιτυχίας απλά αυξάνουμε το amount. Σε περίπτωση αποτυχίας παίρνουμε τον κατάλληλο κόμβο (από την αναζήτηση) και εισάγουμε το νέο string δεξιά ή αριστερά με amount 1.

Καταστροφή

Ξεκινώντας από τη ρίζα, διαπερνάμε το δέντρο με post-order και διαγράφουμε, έτσι ώστε να έχουμε διαγράψει όλους τους απογόνους ενός κόμβου, πριν διαγράψουμε τον ίδιο.

5.Δέντρο AVL

Αναφερόμαστε σε κώδικα από τα αρχεία:AVL.h, AVL.cpp, AVLTree.h και AVLTree.cpp

Όπως και πριν θέλουμε ο προγραμματιστής να αλληλοεπιδρά με το σύνολο, όμως δεν μπορούμε να το κάνουμε όπως στην προηγούμενη κλάση, καθώς η ρίζα μπορεί να αλλάξει θέση. Έτσι «χωρίζουμε» τη δομή σε δύο κλάσεις. Η AVL αντιπροσωπεύει τους κόμβους του δέντρου και υλοποιεί τις λειτουργείες που επιδρούν σε ατομικούς κόμβους (περιστροφές, υπολογισμός ύψους κλπ.), ενώ η AVLTree υλοποιεί τις *συνολικές* λειτουργείες (αναζήτηση, εισαγωγή). Μέσω της AVLTree κρατάμε έναν pointer στη ρίζα του δέντρου, και έτσι όταν, κατά την εισαγωγή δεδομένων, αυτή αλλάζει, μπορούμε να το ανιχνεύουμε και να

ενημερώνουμε τον pointer, χωρίς να επηρεάζουμε τη δομή του δέντρου. Για κάθε λειτουργεία θα αναφέρουμε σε ποια κλάση υλοποιείται.

Η κλάση AVL αποτελείται από pointers σε AVL (parent, left, right), ένα int height και φυσικά, string data και int amount.

Η AVLTree έχει μοναδική μεταβλητή ένα pointer σε AVL που ονομάζεται root.

Βοηθητικές Λειτουργείες

Στην κλάση AVL χρειαζόμαστε μερικές συναρτήσεις για να υπολογίζουμε γρήγορα κάποιες απαραίτητες τιμές.

Η συνάρτηση getHeight() είναι απαραίτητη καθώς δουλεύουμε κυρίως με pointers και πολλές φορές τα pointers αυτά δεν αντιστοιχούν σε ένα πραγματικό αντικείμενο. Έτσι, όταν ζητάμε την μεταβλητή height προκαλούμε σφάλματα. Η getHeight() επιστρέφει τη μεταβλητή height, αλλά όταν καλείται από nullpointer επιστρέφει 0.

Η συνάρτηση updateHeight() καλείται μετά τις εισαγωγές στους απαραίτητους κόμβους των οποίων το ύψος μπορεί να άλλαξε.

Η συνάρτηση getBalanceFactor() επιστρέφει την διαφορά ύψους ανάμεσα στα υπο-δέντρα ενός κόμβου και χρειάζεται για να ανιχνεύουμε ανισορροπίες κατά την εισαγωγή.

Τέλος χρειαζόμαστε συναρτήσεις για τις περιστροφές οι οποίες γίνονται ακριβώς με τον τρόπο που περιγράφεται στο βιβλίο, όπου ο κόμβος για τον οποίο καλείται η συνάρτηση, αντιστοιχεί στον κόμβο x των διαφανειών.

Κατασκευή

Οι κόμβοι κατασκευάζονται με ορίσματα τον κόμβο-γονέα τους και το string data, ενώ τα pointers αρχικοποιούνται ως null και το ύψος και amount ως 1.

Το AVLTree αρχικοποιεί το ριζικό κόμβο ως null και του δίνει πραγματική τιμή μετά την πρώτη εισαγωγή.

Αναζήτηση

Γίνεται με την ίδια μέθοδο που χρησιμοποιεί και το δυαδικό δέντρο. Η ανάγκη για αναδρομή σημαίνει πως η συνάρτηση αναζήτησης υλοποιείται αρχικά μέσα στην κλάση AVL, ενώ από την AVLTree μπορούμε να την καλέσουμε από το root.

Εισαγωγή

Ξεκινάει με μία αναζήτηση. Αν αυτή επιτύχει αυξάνουμε το amount στον κόμβο που επιστρέφει. Διαφορετικά παίρνουμε τον τελευταίο κόμβο που επισκέφθηκε, συγκρίνουμε την νέα μας τιμή με την τιμή του τελευταίου κόμβου και εισάγουμε αριστερά ή δεξιά ανάλογα. Έπειτα καλούμε επαναληπτικά την updateHeight() για όλους τους προγόνους του νέου κόμβου και ελέγχουμε για τυχών ανισορροπίες, τις οποίες αντιμετωπίζουμε με τις κατάλληλες περιστροφές. Η μόνη λεπτομέρεια που πρέπει να προσέξουμε είναι όταν εκτελούμε περιστροφή που επηρεάζει την ρίζα, πρέπει να ενημερώσουμε την μεταβλητή root.

Καταστροφή

Και πάλι χρησιμοποιούμε post-order διαπέραση για να διαγράψουμε πρώτα τους απογόνους ενός κόμβου και μετά τον ίδιο. Η υλοποίηση αυτής της λειτουργείας γίνεται στην AVL λόγω της ανάγκης για αναδρομή. Καλούμε τη συνάρτηση στη ρίζα του δέντρου μέσω της AVLTree και έτσι διαγράφουμε όλους τους κόμβους του δέντρου.

6.Πίνακας Κατακερματισμού

Αναφερόμαστε σε κώδικα από τα αρχεία: HashTable.h και HashTable.cpp

Ορίζουμε κλάση HashTable που (όπως και οι υπόλοιποι πίνακες) αποτελείται από δύο pointers σε string (data) και int (amount) και μία μεταβλητή μεγέθους size.

Κατασκευή

Ο κατασκευαστής χρειάζεται ως όρισμα μόνο το μέγεθος και δεσμεύει την αντίστοιχη μνήμη για τους data και amount. Τα amount αρχικοποιούνται ως 0.

Συνάρτηση Κατακερματισμού

Η συνάρτηση hash() υλοποιείται στην αρχή του αρχείου HashTable.cpp και δέχεται ως όρισμα ένα string, έναν ακέραιο x που συμβολίζει την x-στη προσπάθεια και πρέπει να αυξάνεται για κάθε σύγκρουση, και το μέγεθος του πίνακα. Στην παρούσα υλοποίηση τετραγωνική, αλλά είναι εύκολο να αλλάξει από τον προγραμματιστή για καλύτερες αποδόσεις.

Αναζήτηση

Όπως περιγράφεται και στη θεωρία, βρίσκουμε την «σωστή» θέση μέσω της συνάρτησης hash και τσεκάρουμε για εγκυρότητα (αρκεί να τσεκάρουμε αν το amount είναι 0 για κενή θέση) και σε περίπτωση συγκρούσεων επαναλαμβάνουμε την hash με αυξημένο x.

Εισαγωγή

Αναζητούμε αρχικά την τιμή που δόθηκε και για επιτυχή αναζήτηση αυξάνουμε το amount. Αν η αναζήτηση αποτύχει, τρέχουμε πάλι επαναληπτικά την hash με

το δεδομένο μας μέχρι να βρούμε amount 0 και εισάγουμε εκεί το νέο μας στοιχείο με amount 1.

Καταστροφή

Απλά αποδεσμεύουμε τη μνήμη που δεσμεύτηκε κατά την κατασκευή.

Μία Σημείωση

Έγινε γρήγορα κατανοητό πως δοκιμές με το αρχείο gutenberg.txt θα ήταν αδύνατες. Όλο το πρόγραμμα έχει ελεγχθεί για το πιο αποδεκτό, small-file.txt. Έτσι είναι αδύνατο να γνωρίζουμε με σιγουριά πώς θα αντιδράσει στην εκτέλεση με παρόμοια, μεγάλα αρχεία. Παρόλα αυτά προσπαθήσαμε να σιγουρευτούμε πως το κείμενο χωράει τουλάχιστον στη μνήμη. Η δομή του αταξινόμητου πίνακα (με σταθερό κόστος εισαγωγής) ήταν η μόνη που πέρασε το στάδιο των εισαγωγών. Αφού θεωρητικά ο αταξινόμητος πίνακας σε εκείνη τη φάση του προγράμματος (πριν συμπιεστεί) είναι και η χειρότερη, από άποψη χωρικού κόστους δομή, οι υπόλοιπες δομές θα πρέπει να τα καταφέρουν μια χαρά.

Επίσης οι μετρήσεις χρόνου γίνονται σε millisecond που ήταν η μεγαλύτερη μονάδα με τη δυνατότητα να ανιχνεύσει όλες μας τις μετρήσεις. Παρόλα αυτά στο κανονικό πρόγραμμα αυτές οι μετρήσεις θα τρέχουν για πολύ παραπάνω. Σε 10 ώρες (ένα συντηρητικό κάτω όριο για τη συμπίεση αταξινόμητου πίνακα με είσοδο το gutenberg.txt) υπάρχουν 3.6e+7 millisecond, οπότε είναι πιθανό να εμφανιστούν υπερχειλίσεις και οι μετρήσεις να λένε ότι να ναι. Εμείς δεν γνωρίζουμε κάποιον τρόπο να το αντιμετωπίσουμε αυτό ,πέρα από το να αλλάξουμε τη μονάδα μέτρησης χρόνου στον πηγαίο κώδικα.