ΑΝΑΦΟΡΑ PROJECT ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ 2019

Κατερίνα Δέρβου, ΑΜ:1054185, mail: st1054185@ceid.upatras.gr

Αθηνά Φουσέκη, ΑΜ:1059623 , mail: st1059623@ceid.upatras.gr

Κων/νος Τσιγκούνης, ΑΜ:1059596 , mail: up1059596@upnet.gr

Part I

1. Υλοποιώντας τους τρείς αλγόριθμους με το ίδιο data set παίρνουμε τα εξής αποτελέσματα:

BubbleSort: Total time taken by CPU: 62.43s

InsertionSort: Total time taken by CPU: 27.81s

SelectionSort: Total time taken by CPU: 20.01s

Παρατηρώντας τα αποτελέσματα βλέπουμε ότι ο BubbleSort είναι χαρακτηριστικά πιο αργός από τους άλλους δύο αλγόριθμους, οι οποίοι δεν έχουν πολύ μεγάλη διαφορά στο χρόνο εκτέλεσης μεταξύ τους. Οι πολυπλοκότητες των αλγορίθμων είναι παρόμοιες, με την Selection να έχει Ω(n2)και στην καλύτερη περίπτωση. Οι αλγόριθμοι θεωρητικά έχουν παρόμοιους χρόνους, παρόλα αυτά το BubbleSort εκτελώντας τις περισσότερες συγκρίσεις και swaps, οι οποίες κοστίζουν στον υπολογιστή, έχει με διαφορά το χειρότερο χρόνο εκτέλεσης. Τώρα συγκρίνοντας τούς άλλους δύο αλγόριθμους ο SelectionSort εκτελεί πιο πολλές συγκρίσεις παρά swaps, ενώ ο InsertionSort το αντίθετο. Έτσι ο SelectionSort υπερτερεί σε μικρό βαθμό σε σχέση με τον InsertionSort.

1. Υλοποιώντας τους τρείς αλγόριθμους με το ίδιο data set παίρνουμε τα εξής αποτελέσματα:

MergeSort: Total time taken by CPU: 0.10s

QuickSort: Total time taken by CPU: 0.02s

Παρατηρώντας τα αποτελέσματα βλέπουμε ότι ο QuickSort έχει καλύτερη επίδοση από τον MergeSort, παρόλο που στη χειρότερη περίπτωση του QuickSort θεωρητικά έχουμε Ο(n2). Αυτό συμβαίνει γιατί ο MergeSort γράφει όλα τα στοιχεία σε έναν προσωρινό πίνακα και μετά πάλι πίσω στον αρχικό πίνακα, κάτι το οποίο κοστίζει στον υπολογιστή. Αντίθετα ο QuickSort δεν κάνει καμία περιττή ενέργεια σε περίπτωση που τα στοιχεία είναι στη σωστή θέση.

1. Υλοποιώντας τον Heapsort αλγόριθμο με το ίδιο data set παίρνουμε τα εξής αποτελέσματα:

Heapsort: Total time taken by CPU: 0.04s

Παρατηρούμε ότι ο QuickSort έχει καλύτερα αποτελέσματα από το Heapsort. Αυτό συμβαίνει γιατί για να κάνουμε οποιαδήποτε σύγκριση με το HeapSort πρέπει πρώτα να δημιουργήσουμε το σωρό, κάτι το οποίο κοστίζει. Συμπερασματικά, η θεωρία επαληθεύεται σε μεγάλο βαθμό, παρόλα αυτά δεν λαμβάνει υπόψη το κόστος κάθε βήματος σε υπολογιστική ισχύ. Η θεωρία υποστηρίζει ότι οι καλύτεροι αλγόριθμοι ταξινόμησης είναι ο MergeSort και ο HeapSort, παρόλα αυτά πειραματικά βλέπουμε ότι ο QuickSort υπερτερεί σε σχέση με τους δύο προηγούμενους για τους λόγους που αναφέρθηκαν νωρίτερα.

|  |  |  |  |
| --- | --- | --- | --- |
| Test number: 76538 | **Linear Search** | **Binary Search** | **Interpolation Search** |
| **Time** | 0.000 | 0.000 | 0.000 |
| **Counter** | 15136 | 10 | 15136 |

Παρατηρούμε ότι ο Binary Search έχει τον καλύτερο χρόνο, δηλαδή τις λιγότερες συγκρίσεις, με μεγάλη διαφορά.

Γνωρίζουμε ότι οι αλγόριθμοι έχουν πολυπλοκότητες:

|  |  |  |  |
| --- | --- | --- | --- |
| Searching | AC | BC | WC |
| Linear | O(n) | O(n) | O(n) |
| Binary | O(logn) | O(logn) | O(logn) |
| Interpolation | O(loglogn) | O(loglogn) | O(n) |

Η κατανομή του dataset επηρεάζει την επίδοση του Interpolation Search, καθώς όσο πιο ομοιόμορφα κατανεμημένοι είναι οι αριθμοί, τόσο καλύτερο χρόνο κάνει (O(loglogn)) , ενώ για ανομοιόμορφα δείγματα μπορεί να φτάσει χρόνους γραμμικής αναζήτησης.

1. Προσπαθήσαμε να υλοποιήσουμε τον κώδικα για το BIS και το βελτιωμένο BIS. Ο κώδικας φαίνεται να δουλεύει για συγκεκριμένα νούμερα και όχι για όλα τα 100.000 του data set. Χωρίς κώδικα που να δουλεύει δυστυχώς δεν μπορούμε να παραθέσουμε αποτελέσματα για το συγκεκριμένο αλγόριθμο.

Part II

1. Για την αποτελεσματική ανάγνωση του αρχείου φτιάξαμε ένα struct students το οποίο περιέχει τα AM, name, surname και grade που αντιστοιχούν σε κάθε μαθητή. Επίσης φτιάξαμε ένα ακόμη struct node που αντιπροσωπεύει έναν κόμβο σε δυαδικό δέντρο και περιέχει ένα students (ουσιαστικά το key κάθε κόμβου) και δύο pointers left, right που δείχνουν στα δύο παιδιά του κόμβου. Αρχικά μετά το διάβασμα του αρχείου κρατάμε όλους τους μαθητές σε έναν pointer τύπου students έτσι ώστε να τους ενθέσουμε στη συνέχεια στο δέντρο.

Με τη συνάρτηση insert() ενθέτουμε στο δέντρο, με βάση το ΑΜ τους, τους μαθητές με τη σειρά που βρίσκονται στον pointer. Ύστερα εμφανίζεται το μενού στο χρήστη με τις επιλογές που αναγράφονται στην εκφώνηση.

Η συνάρτηση print\_inorder() εμφανίζει το δέντρο με διατεταγμένη διάσχιση(από το μεγαλύτερο στο μικρότερο) βάσει του ΑΜ τους μαθητές.

Η συνάρτηση search() παίρνει σαν όρισμα ένα ΑΜ και επιστρέφει τον μαθητή στον οποίο ανήκει, έτσι μπορούμε να εκτυπώσουμε όλα τα στοιχεία του συγκεκριμένου μαθητή.

Για την επεξεργασία των στοιχείων ενός μαθητή, χρησιμοποιούμε ξανά τη search(), και στη συνέχεια μπορούμε να επεξεργαστούμε το όνομα, το επίθετο και το βαθμό του επιστρεφόμενου μαθητή από το τερματικό. Για τη διαγραφή μιας εγγραφής φοιτητή χρησιμοποιούμε τη συνάρτηση deleteNode(), η οποία με όρισμα ένα ΑΜ βρίσκει το μαθητή στον οποίο αντιστοιχεί και δρα ανάλογα αν ο κόμβος έχει ένα, δύο η κανένα παιδί. Αν ο κόμβος έχει ένα παιδί αντιγράφει το περιεχόμενο του παιδιού στον πατέρα και στη συνέχεια διαγράφει το παιδί. Αν έχει δύο παιδιά, βρίσκει το διάδοχο του κόμβου, αντιγράφει το περιεχόμενο του διάδοχου στον προκάτοχο και διαγράφει το διάδοχο από το δέντρο. Τέλος, αν ο κόμβος δεν έχει παιδιά, απλά διαγράφεται από το δέντρο.

Όλες αυτές οι επιλογές είναι μέσα σε έναν βρόγχο while με συνθήκη που είναι πάντα αληθής, έτσι αν ο χρήστης θελήσει να αποχωρήσει από αυτή την εφαρμογή με την επιλογή exit γίνεται break στον ατέρμον βρόγχο και η εφαρμογή κλείνει.

1. Με τη δημιουργία του δέντρου με βάσει τους βαθμούς των μαθητών δημιουργήθηκε το θέμα ότι κάποιοι μαθητές είχαν τους ίδιους βαθμούς. Στα δυαδικά δέντρα κανονικά δεν μπορεί να υπάρχει δύο φορές η ίδια τιμή, και αν προσπαθήσουμε να εισάγουμε την ίδια τιμή απλά θα αντικατασταθεί η παλιά κάτι που εμείς δεν θέλουμε. Για να λυθεί αυτό το θέμα δημιουργήσαμε έναν τρίτο pointer μέσα στο struct node τον inside, ο οποίος κρατά τους μαθητές που έχουν ίδιο βαθμό. Θεωρητικά θα μπορούσε κανείς να πει ότι αυτή τη στιγμή έχουμε ένα τριαδικό δέντρο, αλλά πιο πρακτική λύση δεν μπορέσαμε να βρούμε πάνω στο ζήτημα.

Έτσι, φτιάξαμε μια παραλλαγή της προηγούμενης insert(), την insertGrade(), η οποία δημιουργεί το δέντρο με βάσει το βαθμό και αν συναντήσει ίσους βαθμούς βάζει τον τελευταίο μαθητή που εισάχθηκε στο inside του μαθητή με τον ίδιο βαθμό.

Ύστερα εμφανίζεται το μενού που αναφέρει η εκφώνηση. Για την εκτύπωση των μαθητών με το μικρότερο βαθμό, βρίσκεται με τη βοήθεια της συνάρτησης minValueNode() ο μαθητής με το μικρότερο βαθμό και εκτυπώνονται και όσοι μαθητές υπάρχουν και στον inside pointer του συγκεκριμένου μαθητή.

Για την εκτύπωση των μαθητών με το μεγαλύτερο βαθμό γίνεται η αντίστοιχη διαδικασία, αλλά με τη συνάρτηση maxValueNode().

Γ) Για την αποτελεσματική ανάγνωση του αρχείου φτιάξαμε ένα struct students το οποίο περιέχει τα ΑΜ, name, surname και grade που αντιστοιχούν στον κάθε μαθητή. Επίσης φτιάξαμε ένα ακόμη struct node που περιέχει ένα students ( ουσιαστικά το key κάθε κόμβου ) και έναν pointer που θα δείχνει στον επόμενο κόμβο . Αυτό το χρησιμοποιούμε για την linked list ουσιαστικά που θα φτιαχτεί όταν έχουμε ίδιο home bucket. Και τέλος έχουμε και το struct hash το οποίο περιέχει ουσιαστικά τον pointer για το πρώτο στοιχείο της linked list και έναν counter.

Στην main υπάρχει ένα μενού το οποίο έχει τις επιλογές search, edit, delete, display και exit για την αναζήτηση, την τροποποίηση των στοιχείων και την διαγραφή ενός μαθητή. Με την display μπορεί ο χρήστης ανά πασά στιγμή να βλέπει τις εξελίξεις στο hashtable.

Με την συνάρτηση createNode φτιάχνουμε έναν καινούργιο κόμβο με τις πληροφορίες που θέλουμε μέσα.

Με την συνάρτηση add βρίσκουμε το ASCII value των ΑΜ, ώστε να μπορούμε να κάνουμε mod στην insert και να βρούμε σε ποια home buckets θα σταλθούν οι πληροφορίες του κάθε μαθητή.

Με την συνάρτηση insertToHash τοποθετούμε τον κάθε μαθητή αναλόγως με το ΑΜ του στον πίνακα που έχει δημιουργηθεί και αν έχει το ίδιο home bucket με άλλους μαθητές με linked list και counter τοποθετείται στην σωστή θέση.

Με την display εκτυπώνουμε τα περιεχόμενα του hashtable μας.

Η συνάρτηση searchInHash παίρνει σαν όρισμα ένα ΑΜ και επιστρέφει τον μαθητή στον οποίο ανήκει, έτσι μπορούμε να εκτυπώσουμε στην main όλα τα στοιχεία του συγκεκριμένου μαθητή.

Για την επεξεργασία των στοιχείων ενός μαθητή, χρησιμοποιούμε πάλι την searchInHash, και στη συνέχεια μπορούμε να επεξεργαστούμε το όνομα, το επίθετο και τον βαθμό του επιστρεφόμενου μαθητή από το τερματικό.

Για την διαγραφή ενός μαθητή από το hashtable χρησιμοποιούμε την deleteFromHash με όρισμα το ΑΜ του μαθητή που θέλουμε να διαγράψουμε.

Όλες αυτές οι συναρτήσεις βρίσκονται σε έναν βρόγχο while που είναι πάντα αληθής, έτσι αν ο χρήστης θελήσει να αποχωρήσει από την εφαρμογή πρέπει να επιλέξει την exit από το μενού και έτσι γίνεται break στον ατέρμονο βρόγχο με αποτέλεσμα να κλείσει η εφαρμογή.

\*(δεν καταφέραμε να βάλουμε το Hashing μαζί με τον κώδικα του υπόλοιπου δεύτερου μέρους, δεν μπορέσαμε να εντοπίσουμε το πρόβλημα στο να δουλεύουν μαζί. Σας στέλνουμε το Hashing σε ένα ξεχωριστό αρχείο c)