ΑΝΑΦΟΡΑ PROJECT ΔΟΜΩΝ ΔΕΔΟΜΕΝΩΝ

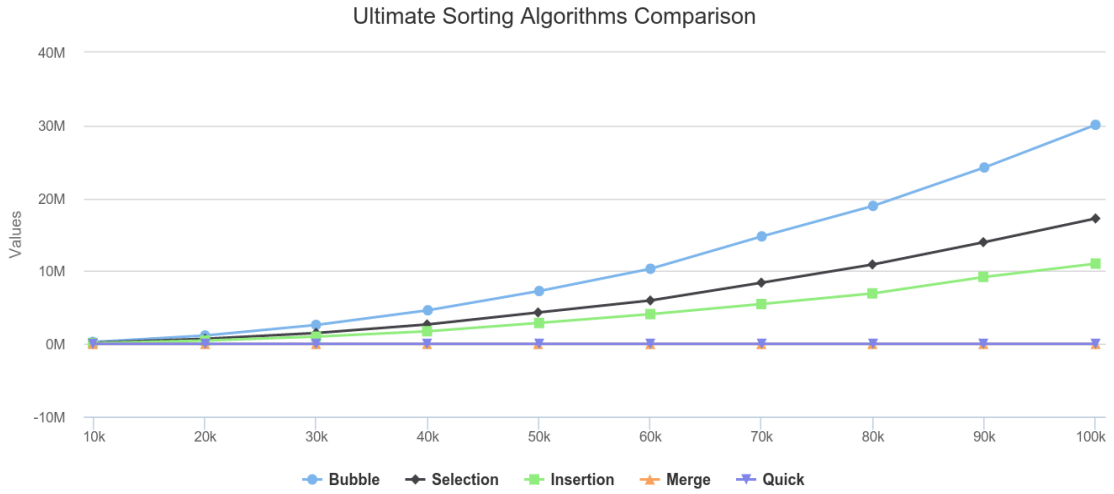
PART 1: "Sorting and Searching Algorithms"

1. Χρησιμοποιώντας ένα τυχαίο data set 100.000 στοιχείων, συγκρίναμε πειραματικά τους αλγόριθμους **Bubble Sort** (Ταξινόμηση Φυσαλίδας), **Insertion Sort** (με Εισαγωγή) και **Selection Sort** (με Επιλογή). Προκειμένου να διαπιστώσουμε τον χρόνο που χρειάστηκε κάθε αλγόριθμος για να τρέξει, χρησιμοποιήσαμε τη συνάρτηση `clock_t` της βιβλιοθήκης της C, `time.h`, η οποία επιστρέφει το χρόνο του επεξεργαστή που έχει καταναλώσει το πρόγραμμα από την αρχή εκτέλεσής του, και έπειτα διαιρέσαμε τον συνολικό χρόνο με `CLOCKS_PER_SEC` για να πάρουμε τον χρόνο σε δευτερόλεπτα.

```
===[Bubble Sort]===
Bubble Time: 32.560164

===[Insertion Sort]===
Insertion Time: 12.946023

===[Selection Sort]===
Selection Time: 14.895800
```



Παρατηρήσεις:
Βλέπουμε πως ο **Bubble Sort** πήρε τον *υπερδιπλάσιο* περίπου χρόνο, σε σύγκριση με τους υπόλοιπους αλγορίθμους, για να τρέξει. Οι **Insertion** και **Selection Sort** χρειάστηκαν παραπλήσιους χρόνους για να τρέξουν, ωστόσο πιο γρήγορος σε θέμα αποδοτικότητας αποδείχθηκε ο **Insertion Sort** (βλέπε διάγραμμα).

2. Διατηρώντας το ίδιο data set με προηγουμένως, συγκρίναμε πειραματικά τους αλγόριθμους **Merge Sort** (Συγχωνευτική Ταξινόμηση) και **Quick Sort** (Ταχυταξινόμηση), λαμβάνοντας τον αντίστοιχο χρόνο που χρειάστηκε κάθε αλγόριθμος για να τρέξει.

```
===[Merge Sort]===
Merge Time: 0.043846

===[Quick Sort]===
Quick Time: 0.049954
```

Παρατηρήσεις:
Βλέπουμε πως και οι δύο αλγόριθμοι πήραν περίπου τον ίδιο χρόνο για να τρέξουν, κι αυτό πιθανότατα οφείλεται στο ότι έχουν την ίδια χρονική πολυπλοκότητα στη μέση περίπτωση, O(nlogn).

3. Διατηρώντας ξανά το ίδιο data set, υλοποίησαμε τον αλγόριθμο του **Heap Sort** (Ταξινόμηση Σωρού), και έπειτα συγκρίναμε τα αποτελέσματα και για τους έξι αλγόριθμους ταξινόμησης που εξετάσαμε μέχρι τώρα.

```
===[Heap Sort]===
Heap Time: 0.040052
```

Παρατηρήσεις:
Βλέπουμε πως ο συνολικός χρόνος του **Heap Sort** είναι παρόμοιος με αυτούς των **Merge**

και **Quick**. Αυτό, για ακόμα μια φορά, οφείλεται στο γεγονός ότι και οι τρεις αυτοί αλγόριθμοι έχουν την ίδια χρονική πολυπλοκότητα στη μέση περίπτωση που εξετάζουμε. Επίσης, συγκρίνοντας πειραματικά αυτούς τους τρεις, με τους υπόλοιπους τρεις (**Bubble**, **Insertion**, **Selection**), παρατηρούμε πως οι **Merge**, **Quick** και **Heap** είναι σαφώς πάρα *πολύ* πιο γρήγοροι από τους άλλους. Αυτή η διαφορά εντοπίζεται στην διαφορετική χρονική πολυπλοκότητα μέσης περίπτωση που ξεχωρίζει την μία τριάδα αλγορίθμων από την άλλη. Πιο συγκεκριμένα, οι **Bubble**, **Insertion**, **Selection** χρειάζονται $O(n^2)$ χρόνο, γεγονός που τους καθιστά πιο αργούς όσον αφορά την απόδοση των **Merge**, **Quick**, **Heap** σε O(nlogn) χρόνο.

4. Χρησιμοποιούμε ένα τυχαίο, ταξινομημένο, data set χωρίς κάποια κατανομή και συγκρίνουμε πειραματικά τους αλγόριθμους **Linear Searching** (Γραμμική Αναζήτηση), **Binary Searching** (Δυαδική Αναζήτηση) και **Interpolation Searching** (Αναζήτηση με Παρεμβολή). Η δειγματοληψία του χρόνου που χρειάζεται για να τρέξει καθένας απ' αυτούς τους αλγόριθμους έγινε με την κλήση της συνάρτησης `clock_t` της βιβλιοθήκης time.h στο «σώμα» της main. Λόγω των πολύ μικρών χρόνων εκτέλεσης, για να βγάλουμε σαφή συμπεράσματα μετρήσαμε κύκλους ρολογιού και όχι χρόνο σε δευτερόλεπτα.

```
Enter the number to search
3920
3920 is present at 814
Linear Time: 9.000000
Enter the number to search
3920
3920 is present at 814
Binary Time: 5.000000
Enter the number to search
3920
3920 is present at 814
Interpolation Time: 7.000000
```

Παρατηρήσεις:

Ως προς τους χρόνους μέσης περίπτωσης, βλέπουμε πως ο **Linear** αλγόριθμος χρειάζεται παραπάνω χρόνο για να βρει ένα στοιχείο. Όσον αφορά τους άλλους δύο αλγόριθμους, ο **Interpolation** υποτίθεται πως έπρεπε να ηταν γρηγορότερος από τον **Binary**, στη μέση περίπτωση, καθώς ο ένας έχει χρονική πολυπλοκότητα O(loglogn), ενώ ο άλλος O(logn). Και αυτό γιατί στον **Interpolation** τα διαστήματα αναζήτησης δεν υποδιπλασιάζονται σε κάθε βήμα (**Binary**), αλλά γίνονται πολύ πιο μικρότερα βάσει του τύπου της παρεμβολής. Στην δικιά μας περίπτωση δεν υπάρχει μεγάλη απόκλιση στους κύκλους ρολογιού, διότι το data set που είχαμε στην διάθεσή μας ήταν μικρό για τα δεδομένα του αλγορίθμου.

5. Υλοποίησαμε τον αλγόριθμο του **Binary Interpolation Search** (BIS), όπως αυτός περιγράφεται στον ψευδοκώδικα του βιβλίου και τον εφαρμόσαμε στην πράξη πάνω στο ίδιο ταξινομημένο data set με προηγουμένως. Ο BIS στη μέση περίπτωση αναζήτησης ενός στοιχείου έχει χρονική πολυπλοκότητα O(loglogn), ενώ στη χειρότερη περίπτωση O(n). Ο κώδικας αδυνατούσε να εντοπίσει τα τελευταία ~100 στοιχεία του dataset, συνεπώς δεν μπορούσε να γίνει μέτρηση για πολυπλοκότητα χειρότερης περίπτωσης. Ίδια δυσκολία προέκυψε και με τον αλγόριθμο της βελτιωμένης έκδοσης του BIS, όπου η ειδοποιός διαφορά του είναι ότι μέσα στο *while loop* το i αυξάνεται εκθετικά (i = 2 * i). Το αναμενόμενο αποτέλεσμα είναι πως ο χρόνος της χειρότερης περίπτωσης της βελτιωμένης έκδοσης θα ήταν μικρότερος από αυτόν της χειρότερης περίπτωσης του BIS, καθώς η χρονική πολυπλοκότητα αλλάζει από $O(\sqrt{n})$ σε O(logn). Η διαφορά των 2 κύκλων ρολογιού ανάμεσα στο BIS και στην βελτιωμένη έκδοση είναι αμελητέα.

```
3920
3920 is present at 814
BIS Time: 6.000000

499076
499076 is present at 99817
BIS Time: 29.000000
```

```
23
23 is present at 2
BIS Time: 5.000000

23
23 is present at 2
BIS_2 Time: 7.000000
```

PART 2: "BSTs && HASHING"

Η εγγραφή κάθε φοιτητή αποθηκεύεται ως `struct` δείκτης με το όνομα `student` (για συντομία `Studentptr`). Στην δομή περιλαμβάνονται τα αλφαριθμητικά ΑΜ, όνομα, επώνυμο και βαθμός. Ο βαθμός αποθηκεύεται και αυτός ως αλφαριθμητικό γιατί ουσιαστικά πρόκειται για ένα στατικό δεδομένο και όχι για έναν μεταβαλλόμενο αριθμό.

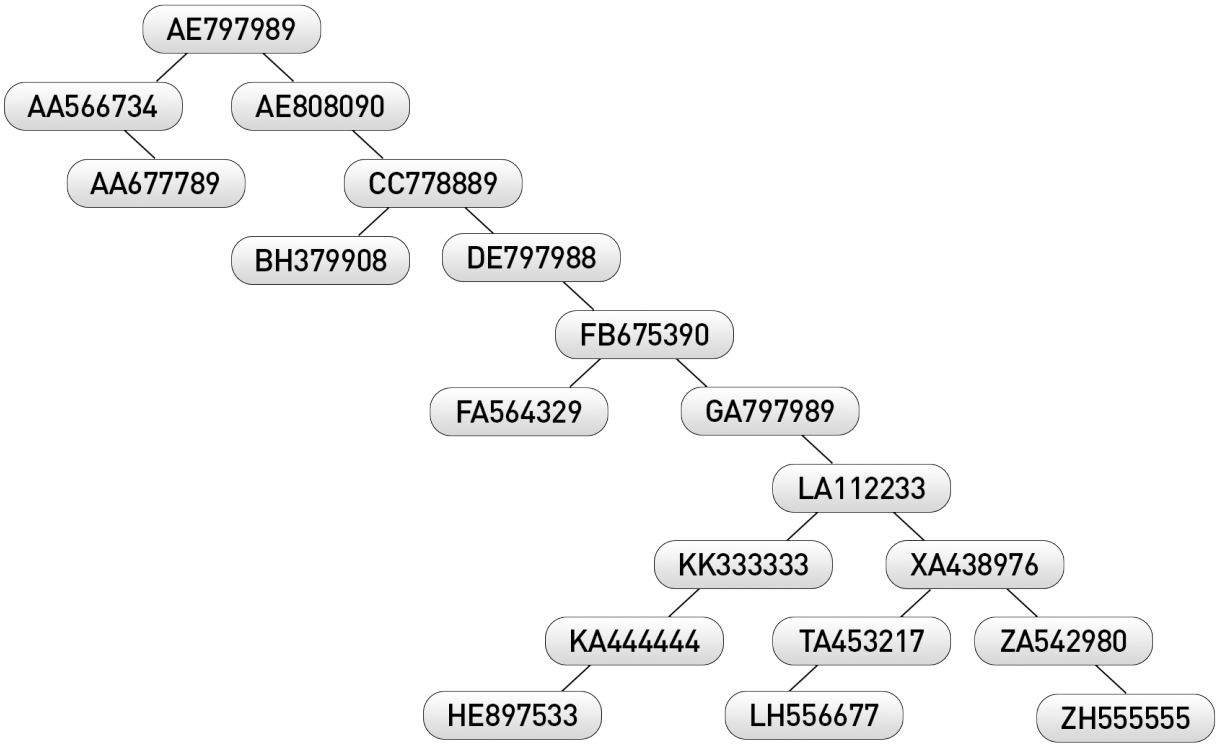
Το πρόγραμμα δίνει επιλογές στον χρήστη ανάλογα με το πως θέλει να φορτώσει το βαθμολόγιο: ως δυαδικό δέντρο ταξινομημένο ανά ΑΜ ή μέσω των βαθμών, ή ως δομή Hashing με αλυσίδες. Ο χρήστης επιλέγει την προτίμησή του μέσω του μενού που του παρουσιάζεται.

Δυαδικό Δέντρο Αναζήτησης με βάση το ΑΜ

Το υποπρόγραμμα έχει ως κύρια συνάρτηση την **`userMenuA()`**. Δημιουργείται η (κενή στην αρχή) ρίζα του δέντρου, η οποία είναι τύπου `Studentptr` μέσω της συνάρτησης **`insertByAM().`** Η ίδια συνάρτηση χρησιμοποιείται για να εισαγάγει καθεμία καινούρια δομή στο δέντρο. Περιλαμβάνει δύο ορίσματα τύπου `Studentptr`: την ρίζα (`root`) και καθεμία καινούρια εγγραφή που της προσθέτουμε (`student`), και επιστρέφει την ρίζα, εκτός από την περίπτωση που δεν έχει δημιουργηθεί ακόμη, που τότε επιστρέφει την καινούρια εγγραφή που θα της προσθέταμε.

Για να προσθέσει μια νέα εγγραφή στο δέντρο συγκρίνει τα αλφαριθμητικά ΑΜ ανάμεσα στο `root` και στο `student` μέσω της **`stringCompare()`**. Η `stringCompare()` συγκρίνει δύο αλφαριθμητικά και επιστρέφει διαφορετικές τιμές ανάλογα με το αποτέλεσμα της σύγκρισης: 1 αν το πρώτο είναι μικρότερο από το δεύτερο, -1 αν είναι μεγαλύτερο και 0 αν είναι ίσα. Η σύγκριση επιτυγχάνεται ελέγχοντας ανά ζεύγη κάθε χαρακτήρα του αλφαριθμητικού· αν όλα τους είναι ίσα, είναι ίσα και τα ίδια τα αλφαριθμητικά, όποτε κάποιο ζευγάρι είναι άνισο,, το αποτέλεσμα της σύγκρισης εκείνων των χαρακτήρων δίνει και την σύγκριση ολόκληρου του αλφαριθμητικού. *Ναι, υπάρχει αντίστοιχη συνάρτηση ήδη στην C, δεν το ξέραμε, τελικά ο χρόνος πράγματι είναι χρήμα, χιχι.*

Η `insertbyAM()` λοιπόν μέσω της `stringCompare()` συμπεράνει ότι το ΑΜ του `student` είναι μικρότερο από την ρίζα, το τοποθετεί αριστερά και σε αντίθετη περίπτωση δεξιά,



σύμφωνα με τον ορισμό του δυαδικού δέντρου, το οποίο επιτυγχάνεται με το να θέτει τα σωστά ορίσματα στους left και right pointers της δομής.

Έτσι πλέον στην `userMenuA()` έχει αρχικοποιηθεί το δέντρο. Μέσω μιας ατέρμονης επανάληψης δίνεται η δυνατότητα στον χρήστη να τροποποιεί συνεχώς το δέντρο χωρίς να σταματάει η εκτέλεση του προγράμματος. Δίνεται οι επιλογές για απεικόνιση του δέντρου, για αναζήτηση ενός συγκεκριμένου φοιτητή μέσω του ΑΜ του, για τροποποίηση των στοιχείων ενός φοιτητή δοθέντος του ΑΜ του και για διαγραφή του.

Στην περίπτωση που θέλουμε να εμφανίσουμε το δέντρο, εκτελείται η **`inorder()`**, η οποία τυπώνει το δέντρο με ενδο-διατεταγμένη διάσχιση. Για την αναζήτηση ενός φοιτητή, στην `userMenuA()` ζητείται το ΑΜ του φοιτητή που περνάει ως όρισμα `chosenAM` στην `searchStudent()`. Η συνάρτηση διασχίζει το δέντρο συγκρίνοντας το ΑΜ τη εκάστοτε ρίζας με το `chosenAM`. Όποτε είναι ίσα, επιστρέφει τη συγκεκριμένη εγγραφή, η οποία αποθηκεύεται σε μια προσωρινή `tempStudent` μεταβλητή, ώστε να εκτυπωθούν τα χαρακτηριστικά της.

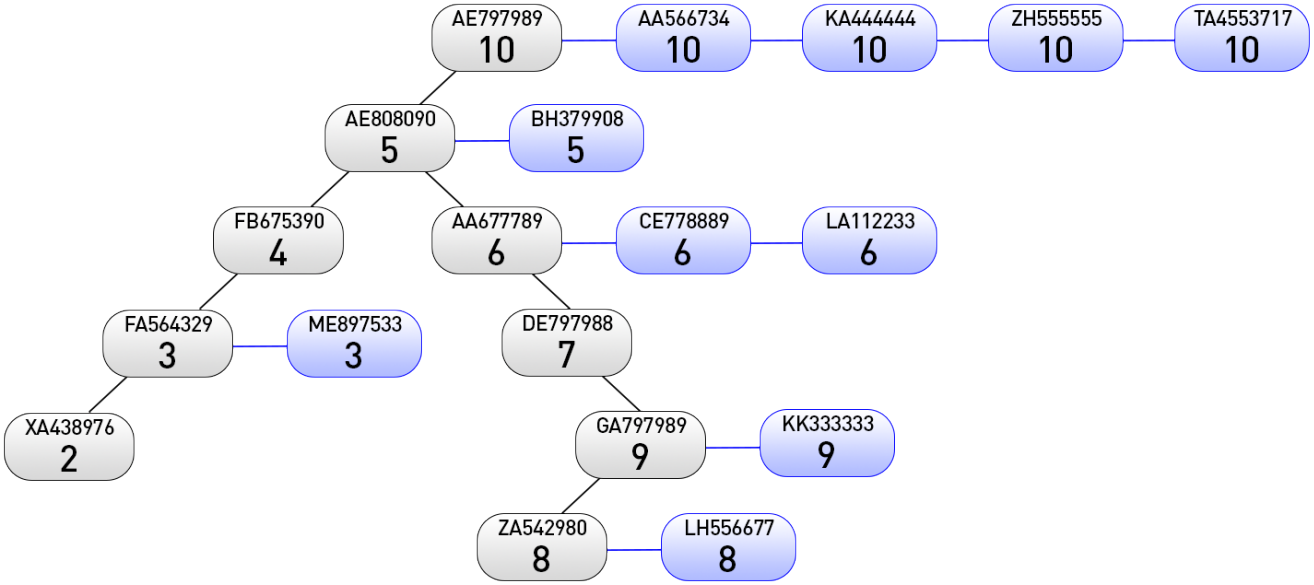
Στην περίπτωση που ο χρήστης θέλει να τροποποιήσει τα στοιχεία ενός φοιτητή, πρέπει να εισαγάγει το ΑΜ του. Εμφανίζεται ένα μενού με επιλογές για τροποποίηση του ονόματος, του επωνύμου και το βαθμού του. Όλα τα ορίσματα μεταφέρονται στην **`editStudent()`** η οποία αφού αναζητήσει τον φοιτητή μέσω της `searchStudent()`, τροποποιεί τη συγκεκριμένη καταχώρηση. Ύστερα ο χρήστη ερωτάται για το αν επιθυμεί να τροποποιήσει κάποιο άλλο πεδίο για τον φοιτητή που επέλεξε ή ακόμα και να δώσει διαφορετικό ΑΜ για κάποιον άλλο φοιτητή. Αυτό επιτυγχάνεται μέσω επαναλήψεων και συνθηκών μέσα στην `userMenuA()`.

Στην περίπτωση διαγραφής ενός χρήστη, το ΑΜ το οποίο θέτει ως όρισμα μεταφέρεται στην **`deleteStudent()`**, η οποία αφού βρει τον ζητούμενο φοιτητή, ελευθερώνει την θέση μνήμης του, και ανάλογα με την θέση του στο δέντρο συνδέει ανάλογα τους γειτονικούς κόμβους του. Φίνα.

Δυαδικό Δέντρο Αναζήτησης με βάση τη βαθμολογία

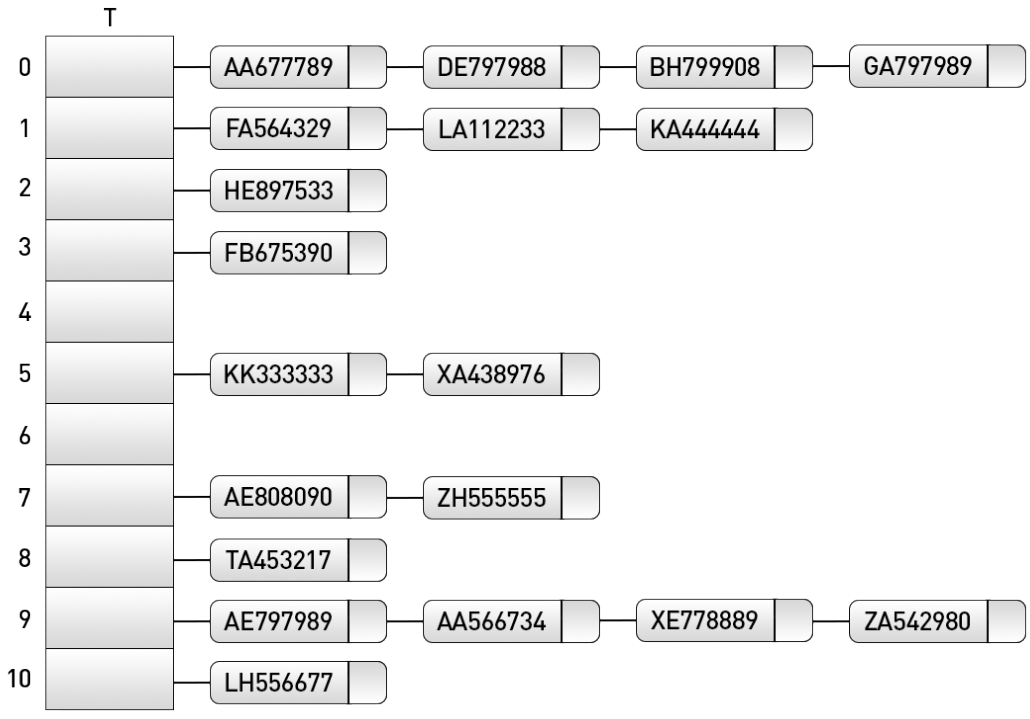
Το υποπρόγραμμα έχει ως κύρια συνάρτηση την **`userMenuB()`**. Με παρόμοιο τρόπο δημιουργείται ο πίνακας με τις δομές των φοιτητών όπως διαβάζονται από το αρχείο. Οι δομές εισάγονται στο δέντρο μέσω της **`insertByGrade()`**, η οποία σε αντίθεση με το (α) ερώτημα πλέον συγκρίνει τα `grade` των φοιτητών. Επειδή οι βαθμολογίες είναι ακέραιοι αριθμοί, αποθηκευμένοι ως strings, μετατρέπονται επί τόπου σε ακεραίους μέσω της `atoi()`. Σε περίπτωση που δύο βαθμολογίες είναι ίσες, κάνουμε χρήση ενός επιπλέον pointer της δομής `struct student`, `sameGrade`. Μέσω αυτού του pointer, αν κάποια βαθμολογία συνυπάρχει ανάμεσα σε δύο ή παραπάνω φοιτητές, οι επιπλέον αυτοί φοιτητές αποθηκεύονται σε μορφή συνδεδεμένης λίστας η οποία συνδέεται με τον αντίστοιχο κόμβο του δέντρου κάθε φορά. Αυτό το μπάλωμα γίνεται γιατί προφανώς είναι άκυρο και εκτός ορισμού να αποθηκεύσουμε duplicate τιμές σε Δυαδικό Δέντρο Αναζήτησης.

Το πρόγραμμα ρωτάει τον χρήστη αν θέλει να εμφανίσει τους φοιτητές με την χειρότερη ή με την καλύτερη βαθμολογία. Ανάλογα με το input του, τρέχει ανάλογη συνάρτηση: `findMaxGrades()` ή την `findMinGrades()`. Και οι δύο συναρτήσεις τρέχουν επαναλαμβανόμενα, γιατί μπορεί την ίδια (κακή ή καλή) βαθμολογία να την έχουν παραπάνω από ένας φοιτητής (δυστυχώς ή ευτυχώς, χιχι). Περιλαμβάνεται δύο static μεταβλητές, ένα `flag` και ένας `counter`. Όσο το `flag = TRUE`, η συνάρτηση αναζητά τον πρώτο (μαύρο) κόμβο με την καλύτερη ή χειρότερη βαθμολογία. Τότε, αφού το επιστέψει στην `userMenuC()`, αρχίζει να προσπελαύνει τους δυνητικούς μπλε κόμβους. Η επανάληψη συνεχίζεται μέχρι ο `sameGrade` να δείχνει σε `NULL`, κάτι το οποίο εξετάζεται στην `userMenuB()`, η οποία είναι αυτή που τυπώνει και τον εκάστοτε φοιτητή.



Δομή Hashing με Αλυσίδες

Ακολουθώντας παρόμοια λογική οργάνωση με το (α), δημιουργούμε τη συνάρτηση **`userMenuC()`**, σε περίπτωση που ο χρήστης επιλέξει τη φόρτωση του αρχείου σε μια δομή Hashing με αλυσίδες. Αρχικά, αρχικοποιούμε τον πίνακα κατακερματισμού (hash table), όπου σε κάθε «κάδο» του αντιστοιχίζεται και μια λίστα, ώστε να είναι κενός. Έπειτα, μέσω της συνάρτησης `insertToHash()`, η οποία είναι τύπου `Studentptr`, κάνουμε εισαγωγή στον πίνακα τις εγγραφές του αρχείου. Στο παρόν σημείο, επισημαίνουμε πως προκειμένου να γίνει επιτυχώς η εισαγωγή των εγγραφών στις σωστές θέσεις του πίνακα, καλείται η συνάρτηση `stringToKey()`. Αυτή η συνάρτηση παίρνει ως όρισμα το ΑΜ της κάθε εγγραφής, και αφού αθροίσει τους κώδικες ASCII των χαρακτήρων του ΑΜ, διαιρεί το αποτέλεσμα με το πλήθος των κάδων (στην προκειμένη περίπτωση είναι 11) και επιστρέφει το υπόλοιπο της διαίρεσης, όπου και είναι η θέση του πίνακα που θα τοποθετηθεί η εγγραφή.



Αφού ολοκληρωθεί η φόρτωση των στοιχείων στον πίνακα, εμφανίζεται στον χρήστη ένα μενού επιλογών, όπου έχει να επιλέξει ανάμεσα στην αναζήτηση ενός φοιτητή βάσει ΑΜ, στη τροποποίηση των στοιχείων ενός φοιτητή βάσει ΑΜ, στη διαγραφή ενός φοιτητή και στην έξοδο του από την εφαρμογή. Σ' αυτό το σημείο, πέρα από αυτές τις επιλογές που μας ζητούσε η εκφώνηση της άσκησης, προσθέσαμε και μια 5η επιλογή, αν ο χρήστης επιθυμεί να δει τα περιεχόμενα, κάθε φορά, του πίνακα κατακερματισμού, κάτι το οποίο είναι αναγκαίο για την επιτυχημένη οπτική σύλληψη των πληροφοριών από τον χρήστη.

Αν ο χρήστης επιλέξει την 1η επιλογή (**Αναζήτηση φοιτητή βάσει του ΑΜ**), καλείται η συνάρτηση `searchInHash()`. Η συνάρτηση αυτή παίρνει ως όρισμα το ΑΜ που έχει ζητηθεί από τον χρήστη να εισάγει στην εφαρμογή και καλώντας κάθε φορά την `stringToKey()` υπολογίζει τη θέση του ΑΜ στον πίνακα και τυπώνει στην οθόνη τις πληροφορίες που συνδέονται με την εγγραφή αυτού του φοιτητή.

Αν ο χρήστης επιλέξει την 2η επιλογή (**Τροποποίηση στοιχείων βάσει ΑΜ**), ερωτάται ποια τιμή των στοιχείων επιθυμεί να αλλάξει, πέραν του ΑΜ, και στη συνέχεια καλείται η συνάρτηση `editInHash()`, η οποία εντοπίζει τη θέση του ΑΜ στον πίνακα και αλλάζει τη τιμή του στοιχείου που έχει επιλέξει ο χρήστης αντικαθιστώντας τη με το `newstring`. Τέλος, επιστρέφει στην οθόνη τη τροποποιημένη εγγραφή του φοιτητή.

Κατά παρόμοιο τρόπο με τις προηγούμενες δύο συναρτήσεις λειτουργεί και η `deleteFromHash()`, όπου αν ο χρήστης επιλέξει την 3η επιλογή, καλείται αυτή η συνάρτηση και αφού εντοπίσει το δοθέν ΑΜ, αποδεσμεύει το χώρο που έπιανε στη μνήμη η αντίστοιχη εγγραφή.

Στοιχεία ομάδας:

Αλέξανδρος Ξιάρχος	1059619	st1059619@ceid.upatras.gr
Νικηφόρος – Γεώργιος Παπαγεωργίου	1059633	st1059633@ceid.upatras.gr