K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού
Περιττοί ΑΜ

- Εαρινό Εξάμηνο 2018-2019
- Διδάσκων: Κώστας Χατζηκοκολάκης
- Εργασία 3
  - Ανακοινώθηκε στις 12 Μαΐου 2019
  - Προθεσμία παράδοσης: 9 Ιουνίου 2019, 12 τα μεσάνυχτα
  - 15% του συνολικού βαθμού στο μάθημα

Πιθανές αντιγραφές

Τα προγράμματα σας θα ελεγχθούν αυτόματα χρησιμοποιώντας κατάλληλο «έξυπνο» λογισμικό για ομοιότητες. Αν υπάρχουν ομοιότητες, **όλη η εργασία μηδενίζεται**. Δεν δίνονται λεπτομέρειες για το πως και το γιατί, απλά ένα στρογγυλό μηδέν (0). Δεν θα έχετε τη δυνατότητα να μιλήσετε με τον καθηγητή ή τους βοηθούς του μαθήματος για περιπτώσεις αντιγραφής.

Κύριο πρόγραμμα

Σε όλες τις παρακάτω ασκήσεις ζητείται η υλοποίηση συνάρτησεων που έχουν κάποια λειτουργικότητα. Θα πρέπει μαζί με τη συνάρτηση αυτή να υλοποιήσετε και ένα κύριο πρόγραμμα (συνάρτηση main) το οποίο θα επιδεικνύει τη λειτουργικότητα **όλων των συναρτήσεων** για κατάλληλα επιλεγμένες εισόδους, και θα πείθει τον βαθμολογητή ότι οι συναρτήσεις λειτουργούν σωστά σε όλες τις ενδιαφέρουσες περιπτώσεις, ώστε να σας βαθμολογήσει με τον υψηλότερο δυνατό βαθμό. Ειναι προτιμότερο η main **να ΜΗΝ δέχεται είσοδο από το χρήστη**, αλλά να χρησιμοποιεί κατάλληλες προκαθορισμένες εισόδους (ώστε ο διορθωτής να μπορεί να την εκτελέσει εύκολα). Επίσης είναι χρήσιμο η main να τυπώνει μηνύματα που να εξηγούν το τί κάνει.

Πως να παραδώσετε τις λύσεις σας

Οι λύσεις θα πρέπει να σταλούν στο e-mail `ddproj@di.uoa.gr` μέχρι την προθεσμία παράδοσης και να είναι οργανωμένες ως εξής. Θα στείλετε **ακριβώς ένα συμπιεσμένο αρχείο**.

- Για κάθε άσκηση θα πρέπει να υπάρχει ένα ξεχωριστό directory με τον main κώδικα της άσκησης (σπασμένο σε περισσότερα από ένα αρχεία .c αν προτιμάτε), και ένα **Makefile** για τη μεταγλώττιση
- Modules κοινά για περισσότερες από μία ασκήσεις θα πρέπει να είναι σε ξεχωριστό directory, πχ ακολουθώντας τη δομή στο demo project που έχει δοθεί στο piazza. Για κάθε module θα υπάρχει μία έκδοση, όχι ξεχωριστές κόπιες για κάθε άσκηση.

Θα πρέπει να υπάρχει και ένα **αρχείο pdf** το οποίο θα περιέχει **όσο documentation χρειάζεται** ώστε οι βαθμολογητές να κατανοήσουν πλήρως τις λύσεις σας και να τις βαθμολογήσουν ανάλογα.

Παρατηρήσεις

- Οπωσδήποτε παρακολουθείτε το piazza για διευκρινήσεις, μικροαλλαγές στην εκφώνηση, κλπ.
- Οι λύσεις σας για όλες τις ασκήσεις πρέπει να είναι οργανωμένες σε modules της C όπως έχουμε συζητήσει στο μάθημα. Για κάθε module απαιτούνται τουλάχιστον 2 αρχεία, `module.h, module.c` και όπου χρειάζεται και ένα `moduleTypes.h`, όπως έχουμε δει στο μάθημα.
- **Δεν χρειάζεται** να ελέγχετε παντού ότι τα ορίσματα μιας συνάρτησης είναι σύμφωνα με τις προδιαγραφές. Ενας τέτοιος έλεγχος μπορεί να είναι χρονοβόρος και άσκοπος (αν πχ η συνάρτηση καλείται μόνο από άλλες συναρτήσεις, όχι απ'ευθείας από το χρήστη, και πάντα με σωστό τρόπο). Σε τέτοιου είδους συναρτήσεις η σύμβαση συνήθως είναι ότι αν η είσοδος είναι λάθος το αποτέλεσμα είναι απρόβλεπτο.
- Συχνά πάλι είναι χρήσιμο να υπάρχουν τέτοιοι έλεγχοι, πχ όταν: τα δεδομένα έρχονται απ'ευθείας από το χρήστη, ο έλεγχος είναι εύκολος, γίνεται μια φορά μόνο στην αρχή, κλπ. Προσθέστε ελεύθερα ελέγχους όπου νομίζετε ότι είναι κατάλληλοι.
- Τα προγράμματα σας θα πρέπει να είναι όσο πιο καλά οργανωμένα γίνεται, σύμφωνα με όσα έχετε μάθει στο μάθημα «Εισαγωγή στον Προγραμματισμό».
- Χρησιμοποιήστε κατανοητά ονόματα μεταβλητών και προσθέστε σχόλια όπου χρειάζεται. Θυμηθείτε: όταν προγραμματίζουμε δεν μας ενδιαφέρει μόνο να τρέχει σωστά το πρόγραμμα, αλλά και να μπορεί να γίνει κατανοητό από κάποιον που το διαβάζει (το διορθωτή, στην προκειμένη περίπτωση)
- Τα προγράμματα σας πρέπει να «τρέχουν» στους υπολογιστές του Τμήματος με το λειτουργικό linux αφού μεταφραστούν με τον μεταγλωττιστή gcc.

Καλή Επιτυχία!

Ασκηση 1 (25 μονάδες)

Υλοποιήστε τον **αφαιρετικό τύπο δεδομένων HashTable** (πίνακας κατακερματισμού) με τις ακόλουθες μεθόδους:

- HTCreate(): creates an empty hash
- HTSize(hash): returns the number of elements in the hash
- HTGet(hash, key, pitem): searches for key in the hash. If found, returns true and copies the item in the pitem pointer passed as argument.
- HTInsert(hash, key, item): inserts a key (with its item) in the hash (replace if exists)
- HTRemove(hash, key): removes the key from the hash
- HTVisit(hash, visit) : calls visit(hash, key, item) for all entries in the hash (in arbitrary order). visit is a function given by the user.
- HTDestroy(hash): destroys the hash by freeing all reserved memory.

Η υλοποίηση πρέπει να χρησιμοποιεί separate chaining. Ολόκληρο το hash table θα πρέπει να αντιπροσωπεύεται από τον τύπο **HTHash**. Τα keys θα είναι πάντα συμβολοσειρές (char*), ενώ τα items θα έχουν έναν αυθαίρετο τύπο **HTItem** που τον δίνει ο χρήστης (όπως στις προηγούμενες εργασίες). Η συνάρτηση κατακερματισμού θα πρέπει να είναι μία κατάλληλη για συμβολοσειρές από αυτές που έχουμε δει στο μάθημα.

Το hash table θα πρέπει να ξεκινάει από ένα προκαθορισμένο μέγεθος, και να μεγαλώνει αυτόματα όταν το load factor ξεπεράσει ένα όριο (δείτε τις διαφάνειες του μαθήματος). Στην περίπτωση αυτή θα πρέπει να διπλασιάζετε το μέγεθος του πίνακα και να προσθέτετε ξανά όλα τα στοιχεία.

Bonus (20 μονάδες): υλοποίηση open addressing με conditional compilation (`make HT_IMPL=separate_chain|open_address`).

Ασκηση 2 (25 μονάδες)

Υλοποιήστε τον **αφαιρετικό τύπο δεδομένων UndirectedGraph** με τις ακόλουθες μεθόδους:

- UGCreate(): creates an empty graph
- UGAddVertex(graph, vertex): adds a vertex
- UGRemoveVertex(graph, vertex): removes a vertex (and all incident edges)
- UGAddEdge(graph, vertex1, vertex2): adds an edge between two vertices
- UGRemoveEdge(graph, vertex1, vertex2): removes an edge between two vertices
- UGGetAdjacent(graph, vertex): returns a list of vertices adjacent to the given one
- UGShortestPath(hash, vertex1, vertex2): returns the shortest path (list of vertices) between the given ones using Dijkstra's algorithm. Returns an empty list if there is no such path.

- UGDestroy(graph): destroys the graph by freeing all reserved memory.

Η υλοποίηση πρέπει να χρησιμοποιεί λίστες γειτνίασης. Ολόκληρος ο γράφος πρέπει να αντιπροσωπεύεται από τον τύπο **UGGraph**.

Οι κορυφές θα πρέπει να είναι **συμβολοσειρές** (τύπος char*). Δηλαδή θα μπορεί ο χρήστης να φτιάξει δύο κορυφές ”foo”, ”bar”, και μια ακμή ανάμεσά τους καλώντας

```
UGAddVertex(graph, ”foo”);
UGAddVertex(graph, ”bar”);
UGAddEdge(graph, ”foo”, ”bar”);
```

Η UGAddEdge θα πρέπει να αντιστοιχεί με αποδοτικό τρόπο τα ”foo”,”bar” στις αντίστοιχες λίστες γειτνίασης.

Ασκηση 3 (25 μονάδες)

Η άσκηση 6 της εργασίας 1 μπορεί να λυθεί εύκολα φτιάχνοντας ένα γράφο όπου οι κορυφές είναι λέξεις, και υπάρχουν ακμές μόνο μεταξύ λέξεων που διαφέρουν κατά ακριβώς ένα γράμμα. Τότε η βέλτιστη μετατροπή είναι το συντομότερο μονοπάτι μεταξύ των δύο λέξεων.

Υλοποιήστε τη λύση αυτή χρησιμοποιώντας το UndirectedGraph module. Αν έχετε υλοποιήσει τις ασκήσεις 6/7 της εργασίας 1, συγκρίνετε το χρόνο εκτέλεσης των τριών μεθόδων.

Ασκηση 4 (25 μονάδες)

Στην άσκηση αυτή θα υλοποιήσετε μια εφαρμογή η οποία θα διαχειρίζεται συναλλαγές για “bitcoins”. Κάθε bitcoin έχει ένα id (string), έχει μία αξία σε $, και μπορεί να ανήκει τμηματικά σε πολλούς χρήστες. Ο κάθε χρήστης αναγνωρίζεται από ένα id (επίσης string) και έχει στην ιδιοκτησία του τμήματα από ένα ή περισσότερα bitcoins. Πχ ο χρήστης ”A” μπορεί να έχει 10$ από το bitcoin ”1”, 20$ από το bitcoin ”2”, κλπ. Ολες οι αξίες σε $ είναι ακέραιοι αριθμοί.

Οι συναλλαγές που πραγματοποιούνται στο σύστημα αφορούν την αποστολή ενός ποσού σε $ από έναν χρήστη σε έναν άλλο. Πχ ο Α μπορεί να στείλει $50 στον Β. Η αποστολή πραγματοποιείται μεταφέροντας στον Β τμήματα ενός ή περισσοτέρων bitcoins που έχει στην κατοχή του ο Α. Πχ μπορεί να μεταφέρει $30 από το bitcoin 1 και $20 από το bitcoin 2.

Οι δομές που θα πρέπει να δημιουργεί το πρόγραμμά σας είναι οι εξής:

- Η βασική δομή είναι ένα **δυαδικό δέντρο για κάθε bitcoin**, κάθε **κόμβος** του οποίου περιέχει ένα struct `BitcoinPart` με πεδία
  - `userId`: ο ιδιοκτήτης του τμήματος
  - `bitcoinId`: το bitcoin στο οποίο αναφέρεται το τμήμα
  - `amount`: η αξία σε $ του τμήματος.

  Η ερμηνεία ενός κόμβου, με δεδομένα πχ (”A”,”1”,$50), εξαρτάται από τη θέση του στο δέντρο:

  - Αν είναι **φύλλο**, σημαίνει ότι την τρέχουσα στιγμή ο Α έχει 50$ από το bitcoin 1. (το άθροισμα όλων των φύλλων είναι πάντα ίσο με την αρχική αξία του bitcoin).
  - Αν είναι **εσωτερικός κόμβος**, σημαίνει ότι στο παρελθόν ο ο Α έχει 50$ από το bitcoin 1 (τότε ο κόμβος ήταν φύλλο), αλλά το τμήμα αυτό συμμετείχε σε ένα transaction (οπότε προστέθηκαν παιδιά στον κόμβο για να καταγραφεί η μεταφορά). Στην περίπτωση αυτή το **αριστερό παιδί** δείχνει τον **παραλήπτη** της μεταφοράς και το **δεξί παιδί** το υπόλοιπο που παραμένει στο χρήστη. Πχ τα παιδιά ("B","1",$30),("A","1",$20) δείχνουν ότι από τα $50, τα $30 μεταφέρθηκαν στον Β, ενώ τα $20 παρέμειναν στον Α. (τα άθροισμα των δύο παιδιών είναι πάντα ίσο με την αξία του πατέρα).

- Το **wallet ενός χρήστη** είναι απλά μία **λίστα από φύλλα** από τα παραπάνω δέντρα. Από τη λίστα αυτή μπορούμε εύκολα να βρούμε τα τμήματα των bitcoins που ανήκουν στον χρήστη την τρέχουσα στιγμή.

- Ενα **transaction** έχει ένα id (int), μια ημερομηνία (long int), και μία **λίστα από εσωτερικούς κόμβους** από τα παραπάνω δέντρα. Από τη λίστα αυτή μπορούμε εύκολα να βρούμε τα τμήματα των bitcoins που συμμετείχαν στο transaction.

- Επίσης θα πρέπει να δημιουργείτε οποιεσδήποτε άλλες δομές σας είναι χρήσιμες για τις συναρτήσεις που υλοποιείτε. Πχ θα χρειαστείτε ένα **hash table** που αντιστοιχεί κάθε user id σε μια λίστα από τα transactions στα οποία ο χρήστης είναι **αποστολέας**, ένα παρόμοιο για τα transactions στα οποία ο χρήστης είναι **παραλήπτης**, και ένα που αντιστοιχεί κάθε bitcoinId στο **αντίστοιχο δέντρο**.

Το δέντρο θα πρέπει να υλοποιηθεί χρησιμοποιώντας το module που φτιάξατε στην εργασία 2, και το hash table με το module που υλοποιήσατε στην Άσκηση 1.

Το πρόγραμμά σας θα πρέπει να υλοποιεί τις παραπάνω δομές, και τις παρακάτω συναρτήσεις που ενεργούν πάνω στις δομές αυτές.

- Initialize : αρχικοποιεί όλες τις δομές του προγράμματός σας

- createUser(userId) : δημιουργεί έναν χρήστη με κενό wallet

- createBitcoin(bitcoinId, userId, amount) : δημιουργεί ένα bitcoin με αξία $amount, το οποίο ανήκει εξ'ολοκλήρου στο χρήστη userId.

- transfer(fromUserId, toUserId, amount) : δημιουργεί ένα νέο transaction (με τωρινή ημερομηνία) που μεταφέρει $amount από τον πρώτο στον δεύτερο χρήστη. Επιστρέφει true αν το transaction πετύχει, και false αν ο χρήστης δεν έχει αρκετά $.

- getUserAmount(userId) : επιστρέφει το συνολικό ποσό (σε $) που έχει στην κατοχή του αυτή τη στιγμή ο συγκεκριμένος χρήστης

- printSent(userId) : τυπώνει όλα τα transactions στα οποία ο χρήστης είναι αποστολέας. Για το καθένα εμφανίζεται ο αποστολέας, ο παραλήπτης και το συνολικό ποσό, πχ

```
A -> B $10
A -> C $20
```

- printReceived(userId) : τυπώνει όλα τα transactions στα οποία ο χρήστης είναι παραλήπτης. Για το καθένα εμφανίζεται ο αποστολέας, ο παραλήπτης και το συνολικό ποσό, πχ

```
C -> A  $10
B -> A  $20
```

- printBitcoinOwners(bitcoinId) : τυπώνει τους τρέχοντες ιδιοκτήτες του bitcoin. Εμφανίζονται μόνο οι χρήστες με μη-μηδενικό ποσό, και μία φορά ο καθένας, πχ

```
A, $20
B, $40
C, $10
```

- printBitcoinHistory(bitcoinId) : τυπώνει όλες τις αλλαγές ιδιοκτήτη τμημάτων του bitcoin (ανεξαρτήτως transaction), πχ

```
A -> B  $10
A -> C  $10
A -> C  $40
```

- Terminate() : ελευθερώνει όλη τη μνήμη

Σημείωση: η άσκηση αυτή περιλαμβάνει (αλλαγμένες) κάποιες από τις βασικές λειτουργίες της φετινής Εργασίας 1 του Προγραμματισμού Συστήματος. Θα διαπιστώσετε ότι, έχοντας υλοποιήσει με γενικό και οργανωμένο τρόπο τις διάφορες δομές που χρειάζεστε, μια τέτοια εργασία βγαίνει αρκετά εύκολα, καθώς όλη η πολυπλοκότητα των δομών κρύβεται πίσω από τα αντίστοιχα modules. Αν έχετε υλοποιήσει έστω και μερικώς την άσκηση αυτή, είστε πλήρως προετοιμασμένοι για ένα από τα πιο απαιτητικά προγραμματιστικά μαθήματα της σχολής!