Κωνσταντίνος Σκορδούλης
ΑΜ : 1115 2016 00155

Η εργασία έχει υλοποιηθεί σε C++ (and compiled with g++), βέβαια τα περισσότερα γίνονται με συναρτήσεις/βιβλιοθήκες της C .
Πολύ χρήσιμη ήταν η **snprintf()** για μετατροπή οποιουδήποτε τύπου σε char* και ακριβή δέσμευση μνήμης με length = snprintf(NULL,0, "%typeofdata", data) +1; (για το '/0') .

Επιπλέον, πολύ χρήσιμη ήταν και η **memcpy(),** μου επέτρεπε να διαβάσω συγκεκριμένο αριθμό bytes από μια διεύθυνση και τα «αναθέτει» σε μία άλλη. Με αυτό τον τρόπο μπορώ να διαβάσω τις εγγραφές **struct Records**(ή **char***) που έχω αποθηκευμένες στο **Bucket**.

Source Files: **bitcoin.cpp(root), Functions.cpp,**

Από **δομές δεδομένων**, χρησιμοποίησα τις απαιτούμενες δομές:

1) Απλή συνδεδεμένη λίστα (**single linked list**) την οποία και έκανα και **template** , λόγω συχνής χρήσης (more on this later) .
2) **HashTables**, τα οποία θα έχουν buckets με bucket_size given from stdio. Χρησιμοποίησα 2 διαφορετικά είδη hash table, class **HashTable,** class **HashTable1** (ίδια λειτουργία, διαφορετικά είδη εγγραφών ➔ explained later).
3) **Binary Tree**, aka Bitcoin Tree, **class Tree**, **class TNode**(TreeNode) (more on this later).

Hash Tables

Στην παρούσα εργασία, δημιούργησα 2 διαφορετικές Hash Table : 1) HashTable, 2)HashTable1.

Το **HashTable** χρησιμοποιείται για τους **senderHT** και **receiverHT**, ενώ το **HashTable1** ➔ έλεγχο εγκυρότητας **transactionID.**

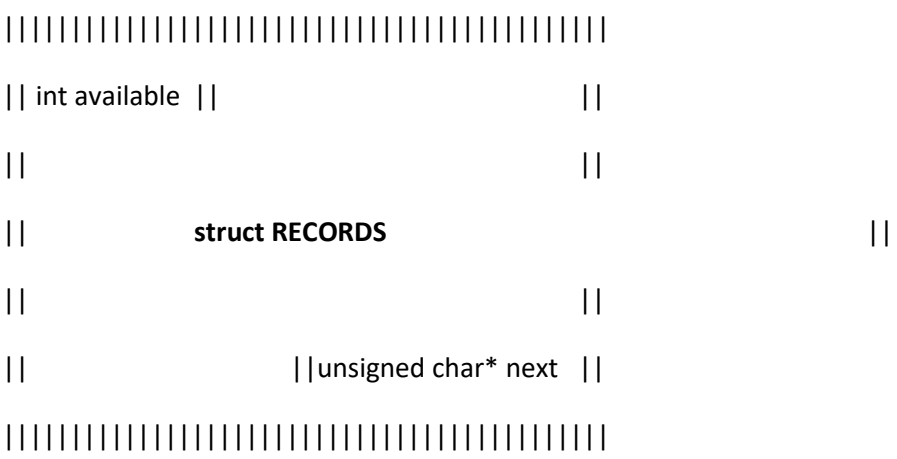
Και οι δύο συναρτήσεις χρησιμοποιούν μια δικιά μου **hash function**, βασισμένη στη universal hash function.

Και οι 2 έχουν την ίδια δομή (όπως περιγράφεται στο header.hpp) , αλλά διαφορετικές εγγραφές στο bucket.

1) Έχει σαν εγγραφή ➔ **struct Records**, η οποία περιέχει **2 δείκτες**:
    a. **Wallet***, δείκτη στη δομή που περιέχει πληροφορίες για το χρήστη/ πορτοφόλι.
    b. **List<UserTLNode>***, δείκτη σε λίστα από κόμβους που περιέχουν δείκτες σε **Transaction** (more on this later).
2) Έχει σαν εγγραφή ➔**char***, δηλαδή τη **διεύθυνση** ενός **string**.

Buckets

Ουσιαστικά είναι **byte arrays** (**unsigned char array[]**), μεγέθους που καθορίζεται από την είσοδο . Η δομή τους (και για τα δυο HashTables) έχει ως εξής :



1) **Int available:** Τα πρώτα **sizeof(int) bytes**, μας δείχνουν πόσος χώρος (εγγραφές) είναι **διαθέσιμος**.
   ***Σημείωση1:** Κατά την εκτέλεση του προγράμματος μπορούμε να υπολογίσουμε **int numOfRecords**, πόσα **Records (maximum)** μπορούν να χωρέσουν σε 1 bucket.
   Άρα το μπορούμε εύκολα να υπολογίσουμε : **current_records = numOfRecords – available**.
2) **Unsigned char* next :** Τα τελευταία **sizeof(unsigned char*) bytes**, περιέχουν την **διεύθυνση** του επόμενου bucket. Αρχικά όλα τα bytes είναι 0 ➔ **next = NULL**.
3) Ο υπόλοιπος (σχεδόν) χώρος είναι για τα **Records (** ή τα **char*** αντίστοιχα).

Bucket Functions

Έχουμε 3 είδη bucket functions : 1) **SearchBucket()**, 2) **Add2Bucket()** , 3) **DeleteBucket()**. Όλες οι συναρτήσεις λειτουργούν **αναδρομικά**

Για να διαβάζω/επεξεργάζομαι τα **Records**(και οποιαδήποτε άλλη μεταβλητή «αποθηκευμένη» στο **bucket**) , δημιουργώ **τοπική** Record και με τη βοήθεια της **memcpy()** , τη "γεμίζω" κάθε φορά με τα bytes, που είναι αποθηκευμένα στο **Bucket**. Εκτελώ κατά κάποιο τρόπο «**ανάθεση**» .

***Σημείωση**: Θεωρώ ότι για να δημιουργηθεί καινούργιο bucket (**overflow bucket**) πρέπει να γεμίσει το προηγούμενο.

1) Ψάχνουμε για ένα συγκεκριμένο **struct Record** .
    a. Βλέπουμε καταρχάς αν **current_records == 0**
        i. **True** ➔ το **Record** δεν υπάρχει (bucket is **empty**).
        ii. **False** ➔ ψάχνω μία μία τις εγγραφές
            1. Αν τη βρω ➔ **return** , found it!
            2. Αλλιώς, τσεκάρω αν **current_records == numOfRecords** (if bucket is full)
                a. **False** ➔ **return,** didn't find it (no overflow bucket)
                b. **True**➔ ελέγχω αν **next == NULL**
                    i. **True** ➔ **return,** didn't find it (no overflow bucket)
                    ii. **False** ➔recursive **call** ➔SearchBucket(**next**)

Σημείωση:** Η παραπάνω συνάρτηση δέχεται σαν όρισμα, **struct Records*, δηλαδή περνάμε τον δείκτη στο struct (**by reference &**), για τον οποίο έχουμε δεσμεύσει χώρο προηγουμένως. Αν βρει το record➔ OK, αλλιώς **delete record; record = NULL;**

2) Με παρόμοιο τρόπο λειτουργεί και η **Add2Bucket**() :
    a. Ελέγχουμε **current_records (= numOfRecords – available)**
        i. If **not full**,τοποθετούμε το record στην άδεια θέση (**with memcpy()**)
        ii. If **full** ➔ ελέγχουμε τον **unsigned char* next**
            1. If **NULL,** δεσμεύουμε χώρο για το καινούργιο bucket και **recursive call** ➔ Add2Bucket(**next**)
            2. Else, recursive **call** ➔Add2Bucket(**next**).
3) Βοηθητική συνάρτηση όταν καλείται ο **destructor** της **HashTable**.
    a. Ελέγχουμε **current_records (= numOfRecords – available)**
    b. Διαγράφουμε 1-1 τα **current_records** . Δηλαδή κάνουμε «ανάθεση» σε 1 τοπική record και τη «**διαγράφουμε**» (**record->wallet = NULL; delete record->list; record->list = NULL;**)
    c. Ελέγχουμε **available == 0** (**full** bucket)
        i. **False** ➔ **Return;**
        ii. **True** ➔ ελέγχουμε αν **next == NULL**
            1. **True** ➔ **Return;**
            2. **False** ➔ Recursive call ➔ DeleteRecord(**next**)

***Σημείωση**: Υπάρχουν και οι συναρτήσεις **SearchBucket1()**, **DeleteBucket1()**, **Add2Bucket1()**

List<template> && Node<template>

Η **class List** αποτελείται από **Node<class T>* head; Node<class T>* tail;**

Η **class Node** αποτελείται από **T* data; Node<class T>* next;**

Ουσιαστικά, αυτό που συμβαίνει είναι:

||||||||||||||||||||||| ||||||||||||||||| |||||||||||||||||

|| template Node || ➔ || **Real** Node || ➔ || **Data** ||

||||||||||||||||||||||| ||||||||||||||||| |||||||||||||||||

Αυτό το κάνω για να μπορώ εύκολα να καθορίσω τι ενέργειες θα γίνονται όταν θα καταστρέφεται ένα **Node**. Βέβαια αυξάνεται λίγο το κόστος …

Είδη Nodes

1) **UserBNode:** User Bitcoin Node. Κάθε χρήστης/**Wallet** έχει μια λίστα από κόμβους που περιέχουν πληροφορίες για την κατοχή bitcoin (520 (bitcoinID) , 5$ (original value 10$), etc.). Κάθε **UserBnode** αντιστοιχεί σε **bitcoin** του χρήστη. Επιπλέον οποιοδήποτε bitcoin αποκτά αξία **0$** (χρησιμοποιείται πλήρως) , αφαιρείται από τη λίστα.
2) **UserTLNode:** User Transaction List Node. Οι κόμβοι που αντιστοιχούν στις λίστες του **ReceiverHashTable**, **SenderHashTable**. Περιέχουν πληροφορίες για τις transactions που συμμετείχε ως receiver και ως sender αντίστοιχα.

   ***Σχεδιαστική Επιλογή**: Για να μειώσουμε το **data-duplication**, έχω σε μία άλλη κεντρική λίστα ΟΛΑ τα transactions, και τα **UserTLNode** περιέχουν απλά **δείκτες** σε αυτά τα transactions.

3) **RLNode:** Root List Node. Κόμβος μιας λίστας που περιέχει (bitcoinID, Tree*) , δηλαδή δείκτη στο αντίστοιχο Bitcoin-Tree.

4) **Wallet**: Περιέχει πληροφορίες για το χρήστη/πορτοφόλι (το **υπόλοιπο**, καθώς και τα **bitcoins** που έχει στη κατοχή του.

5) **PTNode**: Pointer Tree Node. Κόμβος που περιέχει δείκτη σε κόμβο δέντρου.

***Σχεδιαστική Επιλογή**: Τα **PTNode** χρησιμοποιούνται ως **διευκόλυνση**. Δηλαδή βρίσκονται μέσα στα **UserBNode**, και μας δείχνουν **κόμβους** του **BitcoinTree** τους οποίους μπορούμε να «**σπάσουμε**».

6) **TLNode:** Transaction List Node. Κόμβος της λίστας που περιέχει ΟΛΕΣ τις **Transactions.** Κάθε κόμβος περιέχει **δείκτη** σε **class Transaction**. Κάθε **Transaction** περιέχει όλες τις απαραίτητες πληροφορίες για μια transaction (**TransactionID**(char *), **senderID**(char *), **receiverID**(char *), **amount**(int), **timestamp**/(date,time) ➔ time_t .

***Σχεδιαστική Επιλογή**: Το date/time αποφάσισα να το μετατρέψω σε **unix timestamp**, για να είναι πιο εύκολες οι συγκρίσεις/πράξεις. Για αυτό το λόγο πολύ χρήσιμες ήταν οι **strptime()** , **strftime(), localtime(), mktime()** . Αν θέλω να αντλήσω **HH:MM** ή **DD-MM-YYYY** **ΜΟΝΟ** , έπαιρνα το **timestamp** ➔ **struct tm** , και μετά με μια δικιά μου struct tm αντλούσα

tm_hour, **tm_minute**(ή **tm_day, tm_month, tm_year**) και μετά τη δικιά μου struct ➔ **timestamp.**

Bitcoin Tree

Αποτελείται από **TNodes** (Tree Nodes) . Κάθε **TNode** περιέχει (userID, amount, αριστερό παιδί, δεξί παιδί, **TLNode***). Δηλαδή περιέχει δείκτη σε κάποιο **Transaction List Node** (την λίστα που περιέχει ΟΛΑ τα transactions)

***Σχεδιαστική Επιλογή**: Το παραπάνω είναι σχεδιαστική επιλογή που μας διευκολύνει για την **TraceCoin()** αργότερα. Το σκεπτικό έχει ως εξής :

" Όταν γίνεται κάποια συναλλαγή, 1 ή περισσότεροι κόμβοι **σπάνε** σε 2 άλλους κόμβους (από το ίδιο ή διαφορετικά δέντρα). Άρα παράγεται **σίγουρα** το αριστερό και μπορεί να παραχθεί το δεξί παιδί (αν transaction_amount < parent_amount). Δηλαδή το κύριο προϊόν της συναλλαγής είναι το **αριστερό παιδί** .

Για αυτό το λόγο, ΟΛΑ τα **αριστερά παιδιά** έχουν το **TLNode* != NULL** (δείχνοντας στο αντίστοιχο transaction από το οποίο δημιουργήθηκαν). Οι υπόλοιποι **TLNode* = NULL;** . "

Αρχικοποίηση

1) Αρχικοποιούμε τις παραπάνω δομές (σύμφωνα με τα ορίσματα που μας έχουν δώσει).
2) Διαβάζουμε το **BitcoinBalancesFile.txt** (με τη βοήθεια της **getline()**). Θεωρούμε πως το max μέγεθος του **WalletID[50]** και του **BitcoinID[50] .** Πριν εισάγουμε χρήστες στο σύστημα , κάθε φορά ελέγχουμε :
    a. **User exists? ➔ Exit!!** . Διατρέχουμε τη λίστα(**List<Wallet>**) για το αν υπάρχει ίδιο walletID. Δεν μπορούν να υπάρχουν δύο ίδιοι users στο σύστημα.
    b. **Bitcoin exists? ➔** Exit!! . Διατρέχουμε τη λίστα(**List<RLNode>**) για το αν υπάρχει ίδιο bitcoinID. Κατά την αρχικοποίηση, δεν επιτρέπεται δύο χρήστες να έχουν το ίδιο Bitcoin.
    c. Αν όλα **OK!** , τότε δημιουργούμε το χρήστη(μαζί με τα **UserBnodes** του) και τον εισάγουμε στη λίστα μας.
3) Διαβάζουμε το **TransactionsFile.txt .** Θεωρούμε πως το max μέγεθος του **TransactionID[50].** Ελέγχουμε κάθε transaction (πριν το δημιουργήσουμε) :
    a. **TransactionID exists➔ Ignore transaction!!** . Ελέγχουμε αν υπάρχει το συγκεκριμένο transactionID, με τη βοήθεια της **HashTable1* TransactionId;** . Λόγω του μεγάλου πλήθους transactions , εδώ ήταν απαραίτητη η χρήση ενός hash table , αντί να ανατρέξουμε τη λίστα.
    b. **SenderID doesn't exist ➔ Ignore transaction!!** . Ελέγχουμε αν υπάρχει το SenderID , διατρέχοντας τη λίστα.
    c. **SenderID == ReceiverID ➔ Ignore transaction!!.**
    d. **ReceiverID doesn't exist ➔ Ignore transaction**!! .
    e. Τώρα για το **amount** :

i. Amount >= 0 ➔ continue;
ii. Sender->amount >= transaction_amount (else ignore transaction).

Μετά δημιουργούμε το **Transaction**, και ενημερώνουμε τις δομές. Για δικιά μου διευκόλυνση κρατάω 2 βοηθητικές μεταβλητές , **int maxID**, **time_t last_trans**.

Η 1η βασίζεται στο σκεπτικό ότι, εάν μας δοθούν (μεταξύ άλλων)ακέραια TransactionID , τότε μετά την αρχικοποίηση, μπορούμε να αποδίδουμε **maxID+1, maxID+2,** ….. Εννοείται πάντα σε **char*** μορφή. Για να ελέγχουμε εάν ένα TransactionID είναι αριθμός , χρησιμοποιώ την **isNumber()** (my implementation) η οποία χρησιμοποιεί την **isDigit()** (stdlib).

Η 2η μας επιτρέπει να κρατήσουμε σε **unix timestamp** την ημερομηνία και ώρα της τελευταίας συναλλαγής, έτσι ώστε να **απορρίψουμε** στο επόμενο βήμα όποιες συναλλαγές είναι **πριν από** την **last_trans**.

Εντολές

Ο χρήστης μπορεί να πληκτρολογήσει εντολές για να εκτελέσει το σύστημα. Αν δεν αντιστοιχεί σε καμία εντολή , εκτυπώνει λάθος και ο χρήστης ξαναπληκτρολογεί.

1) **RequestTransaction:** Πριν εκτελεστεί κάποια Transaction γίνονται οι γνωστοί έλεγχοι (που προαναφέραμε) +1 **extra** έλεγχος. Δηλαδή να μην προηγούνται της τελευταίας συναλλαγής :
   a. **Timestamp < last_trans ➔ Ignore Transaction!!**.
   b. Αν δεν μας δοθεί **date,time** τότε δεν χρειάζεται ο παραπάνω έλεγχος

   Έπειτα δημιουργούμε το transaction, και ενημερώνουμε τις δομές (καθώς και τις **maxID, last_trans**).

2) RequestTransaction**S** : Σχετικά ίδιο με το παραπάνω.
   a. Ο χρήστης γράφει το transaction, το οποίο πρέπει να τελειώνει με **";"** (γίνεται έλεγχος για αυτό). Επεξεργαζόμαστε το αίτημα του (με τους γνωστούς ελέγχους).
      i. Αν δεν γίνει δεκτό ➔ πληκτρολογεί ο χρήστης καινούργια εντολή
      ii. Αν γίνει δεκτό, τότε εισάγεται στο **"request transactions mode"** (εμφωλευμένη **while(getline())**) κατά την οποία ο χρήστης δεν χρειάζεται να επαναλάβει «**/requestTransactions**» και απλά πληκτρολογεί το **transaction** που θέλει.
         1. Για αυτά τα transactions, γίνονται οι ίδιοι έλεγχοι και αν περάσουν τους ελέγχους ➔ ενημερώνουμε κατάλληλα τις δομές.
         2. Για να βγει από αυτό το mode ➔ "**/exit**" και επιστρέφουμε σε «κανονική» λειτουργία.

b. Ο χρήστης απλά εισάγει σαν όρισμα **inputFile**, το οποίο περιέχει transactions. Διαβάζει μία-μια γραμμή , αν περνάει τους ελέγχους ➔ ενημερώνει κατάλληλα τις δομές.

3) **findEarnings:** Ψάχνουμε το συνολικό κέρδος του χρήστη WalletID , και τις συναλλαγές(που συμμετείχε σαν **receiver**) στο διάστημα που ορίζεται από το input. Εκμεταλλευόμαστε την **HashTable* receiverHT**.
   a. Ελέγχουμε αρχικά αν υπάρχει ο walletID (αν δεν υπάρχει ➔ **Exit**).
   b. Μετά έχουμε 3 σενάρια:
      i. Μας δίνεται **time1**, **time2** (εκτυπώνουμε λάθος αν δεν υπάρχει η time2). Αφού διαμορφώσουμε κατάλληλα τα timestamps, βρίσκουμε τις συναλλαγές που έχουν γίνει σε **ορισμένη ώρα οποιαδήποτε μέρα**.
      ii. Μας δίνεται **date1**, **date2** (εκτυπώνουμε λάθος αν δεν υπάρχει η date2). Αφού διαμορφώσουμε κατάλληλα τα timestamps, βρίσκουμε τις συναλλαγές που έχουν γίνει σε **ορισμένη ημερομηνία οποιαδήποτε ώρα**.
      iii. Δεν μας δίνεται **τίποτα** ➔ εκτυπώνουμε **όλες τις συναλλαγές**.
4) **findPayments:** Ίδιο με τη **findEarnings.** Αυτή τη φορά χρησιμοποιούμε **HashTable* senderHT**.
5) **walletStatus:** Διατρέχουμε την **List<Wallet>** και αν υπάρχει ο χρήστης εκτυπώνουμε **Wallet->amount** (το υπόλοιπο του χρήστη).
6) **bitCoinStatus:** Διατρέχουμε την **List<RLNode>**, για να βρούμε το **Bitcoin Tree** (αν δεν υπάρχει **ignore**). Μετά υπολογίζουμε 1) πόσα **transactions** έχουν υλοποιηθεί και 2) **unspent amount**
   a. Το $1^{ο}$ πετυχαίνεται με την αναδρομική συνάρτηση **TNode::TransCount()**, η οποία υπολογίζει όλα τα **αριστερά παιδιά**. Όμως επειδή υπάρχει περίπτωση 2 ή παραπάνω σπασίματα (στο ίδιο δέντρο) να αντιστοιχούν στην **ίδια transaction**, χρησιμοποιούμε **HashTable1* visited;** , όπου τσεκάρουμε εάν έχουμε ξανά συναντήσει , ένα συγκεκριμένο TransactionID.
   b. Για το $2^{ο}$ θεωρούμε ότι το **πιο δεξιό φύλλο** (αν υπάρχει) αποτελεί το **unspent amount**. Χρησιμοποιούμε την **TNode::Unspent()** (η οποία και αυτή είναι αναδρομική)
7) **traceCoin:** Διατρέχουμε την **List<RLNode>**, για να βρούμε το **Bitcoin Tree** (αν δεν υπάρχει **ignore**). Μετά για κάθε **αριστερό παιδί**: Tnode->node->transaction->Print() . Όπως και παραπάνω , υπάρχει κίνδυνος να εκτυπώσουμε την ίδια συναλλαγή 2 φορές ➔ **HashTable1* visited** , και έτσι τα εκτυπώνουμε μόνο 1 φορά
8) **exit:** Αποδεσμεύουμε καταλληλα τη μνήμη **and exit**.