Introduction

In this assignment you will implement a program that will accept, process, record and answer queries for Bitcoin transactions. In particular, you will implement a set of structures (hash tables, linked lists, trees) that allow the insertion and queries of a large volume of bitCoinTransaction type records. Although the exercise data will come from files, ultimately all records will be stored in main memory only.

Application interface The application will be called bitcoin and will be used as follows: ./bitcoin -a bitCoinBalancesFile –t transactionsFile -v bitCoinValue –h1 senderHashtableNumOfEntries –h2 receiverHashtableNumOfEntries –b bucketSize where:

- The bitCoinValue parameter tells the application what the value (in $) of a bitcoin is. - The senderHashtable1NumOfEntries parameter is the number of positions of a hash table that the application will keep for locating transaction sender information. - The receiverHashtable1NumOfEntries parameter is the number of positions of a hash table that the application will keep for locating transaction receiver information. - The bucketSize parameter is the number of Bytes that gives the size of each bucket in the hash tables.

- The bitCoinBalancesFile (or some other file name) is a file that includes the initial balances of users participating in the bitcoin network. Each line of this file is a list of bitCoin IDs owned by a particular user (userID). For example if the contents of the file are:

My 123 337 880 667
Kylian 456 767 898
Katerina 222 567 003

means that we initially have three users participating in the network. Mia has four bitcoins, with bitCoinIDs 123, 337, 880, 667, Kylian has three bitcoins with IDs 456, 767, 898, coc. You can assume that the username does not contain spaces.

- transactionsFile (or some other file name) is a file containing a series of requests (transactions) to be processed. Each line of this file describes a transaction with the userIDs of the sender and receiver and the amount ($ in $) the sender is sending. For example if the contents of the file are:

889 Maria Ronaldo 50 2018-12-25 20:08
776 Lionel Antonella 150 14-02-2019 10:05

means we have two transactions where Maria asks to send $50 to Ronaldo on 12-25-2018 at 8:08pm and Lionel asks to send Antonella $150 on 2-14-2019 at 10:05am. You can assume that the contents of the transaction file will be sorted by date and time. Specifically, a transaction record/request is a line of ASCII text consisting of the following elements: 1. transactionID: a string (can only have digits) that uniquely specifies each such record.
2. senderWalletID: a string consisting of letters. 3. receiverWalletID: a string consisting of letters. 4. value: the amount of the transaction (you can assume it is an integer).

5. date: date the transaction request is made. It must be in the format DD-MM-YYYY where DD represents the day, MM the month, and YYYY the time of the request.
6. time: time in 24 hours when the transaction request is made. It must be in the format HH:MM where HH represents the hour and MM represents the minute.

To begin with, your application will need to open the bitCoinBalancesFile and transactionsFile files, read one line at a time, and initialize and store in memory the data structures it will use when running queries. You must check that the information in the files is valid. For example, if in the file bitCoinBalancesFile, there are users who own the same bitcoin, you should handle the error by displaying the appropriate message and exiting the application. Also, if while processing the transactionsFile you find a transaction that is not valid, then the application should display a message that the transaction is not possible and is canceled. One such case is when in a transaction, the sender does not have enough money in his wallet to complete the transaction.

When the application finishes processing the bitCoinBalancesFile and transactionsFile files, it will wait for user input from the keyboard. The user will be able to give the following commands:

- /requestTransaction senderWalletID receiverWalletID amount date time

The user requests to send amount money from the user with userID senderWalletID to the user with userID receiverWalletID. The date and time must be later than the last transaction recorded by the application, otherwise the request is rejected. If no date and time have been given, the application uses the current time to record the execution time of the transaction. Also, the application should check if there are sufficient funds to complete the transaction successfully. If there are, it updates the appropriate structures (see below) and presents a success message to the user with the details of the transaction. If there are none, the application presents a failure message to the user.

- /requestTransactions senderWalletID receiverWalletID amount date time;
  senderWalletID2 receiverWalletID2 amount2 date2 time2;
  ...
  senderWalletIDn receiverWalletIDn amountn date time;

The user requests a series of transactions to be performed. Transactions are separated by a Greek semicolon. The application checks the validity of each transaction and accordingly updates the data structures and presents a result message to the user.

- /requestTransactions inputFile

The user requests that transactions described in inputFile be performed. Transactions are separated by a Greek semicolon and have the same format as requestTransaction(s) queries. The application checks the validity of each transaction and accordingly, updates the data structures and presents a result message to the user.

- /findEarnings walletID [time1][year1][time2][year2] The application first returns the total amount received through transactions by the user with userID walletID (selectable in the time and/or date range). If there is a definition for [time1] there should also be a definition for [time2]. The same also applies to the use of the non-mandatory ones

parameters [year1] and [year2]. Then, it shows all the transaction records of the user (as the recipient) that were successfully executed within the specified interval. If no interval is specified, then the application will show the complete transaction history where walletID is a recipient.

- /findPayments walletID [time1][year1][time2][year2] The application returns
the total amount that the user with userID walletID has successfully sent through transactions (selectable in the time and/or date range). It then displays all transaction records of the user (as the sender) that were successfully executed within the interval given on the command line. If no interval is specified, then the application will show the complete transaction history where walletID is the sender.

- /walletStatus walletID The
application returns the current amount in the wallet walletID.

-/bitCoinStatus bitCoinID The
application returns the initial value of the bitcoin me ID bitCoinID, the number of transactions in which it has been used, and the amount of bitCoinID left unspent (ie not yet used in a transaction).

Example output: 124 10 50 means
that bitcoin 124 has been used in 10 transactions while 50 units of its value have not yet been used in a transaction.

- /traceCoin bitCoinID The
application returns the history of transactions involving the bitcoin bitCoinID.

Example output:

/tracecoin 124
889 Maria Ronaldo 50 2018-12-25 20:08
776 Lionel Antonella 150 14-02-2019 10:05

Maria gave Ronaldo 50 credits on 12/25/2018 (via transaction #889) and Lionel gave 150 credits to Antonella on 2/14/2019 (via transaction #776).

-/exit
Exit the application. Make sure you free all allocated memory properly.

Data Structures You
can use C or C++ to implement the application. However, you cannot use the Standard Template Library (STL). All data structures should be implemented by you. Make sure you only allocate as much memory as you need, e.g. the following tactic is not recommended:

int wallets[512];// store up to 512 wallets, but really we don't know how many

Also make sure that you free the memory properly both during the execution of your program and on exit.

To complete the exercise you will need, among other things, to implement the following data structures.

1. Two indexed hash tables (senderHashTable and receiverHashTable) provide fast access to successfully executed transaction data. The first hash table essentially provides access to transaction details where the walletID is a sender. The second table does the opposite: for each walletID it provides access to transaction details where the walletID is a recipient. Hash tables will use buckets to serve walletIDs that present a "collision" (ie, the result of the hash function leads to the same hash table element) that more than one bucket is needed to store data, they are created dynamically and arranged in a list.

2. For each walletID hashed into an element of the senderHashTable, there is a set of transactions in which it is a sender. This set is placed in a dynamically linked list. Each node in the list contains access to elements of a transaction. Corresponding structures should be made with the receiverHashTable.

3. For each walletID, a structure (of your design choice) containing access to wallet elements (such as, for example, the total current balance of the walletID, the bitCoinIDs that the walletID owns, and the rest of them, etc). When the application processes a transaction, it must check if there is enough money in total in the sender's wallet before proceeding to complete the transaction. If present, all appropriate data structures should be properly updated during transaction execution. 4. A tree for each bitCoinID that will hold the entire bitcoin transaction history and will be dynamically created as transactions involving that bitcoin are processed. The root node of the tree will contain the walletID of the original owner of the bitCoinID, and the original value of the bitCoinID. To perform a transaction, if some money is needed from the bitCoinID (from the sender's wallet), the application will add a new node (leaf) to the tree containing the walletID of the recipient who received the money and the amount received. Also, in case there is a balance in the bitcoin after the transfer, a second new node should be added that reflects the balance of bitcoin left with the sender. For example, Figure 1 shows a tree for bitcoinID 123 that starts in A's possession with value 50. When A sends M $20, a new node with elements M and $20 and a new node with elements A and $30 is created, because A has $30 of bitcoin 123 left. In this way, one can traverse the tree of a bitcoin and find the history of transactions involving that particular bitcoin. The sum of the amounts of the leaves of the tree should always be equal to the original value of the bitcoin.

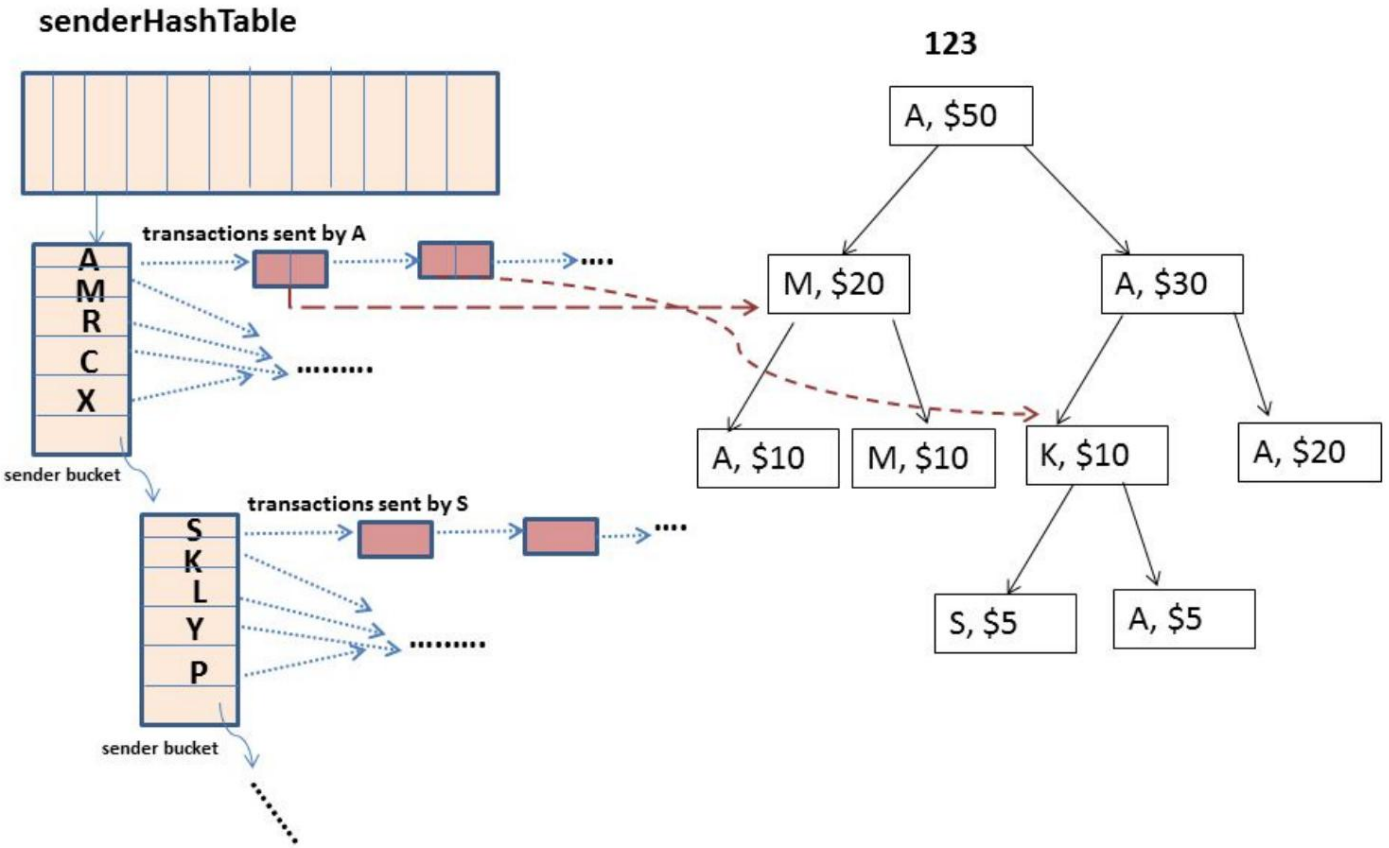

Figure 1: Example of Some Structures for the bitCoin Application

Your goal in this exercise is to have as *little data duplication as possible.* Make sure that for each sub-problem you need to solve when implementing the exercise, you use the most efficient algorithm or data structure. Any design decisions and choices you make during implementation should be described in the README, in the deliverables.

Deliverables

- A short and concise explanation of the choices you've made in designing your program. 1-2 pages of ASCII text is enough. Include the explanation and instructions for compiling and running your program in a README file with the code you submit.

- Any source of information, including code that you may have found on the Internet or elsewhere should be cited in your source code as well as in the README above. - All your work (source code, Makefile and README) in a tar.gz file named OnomaEponymoProject1.tar.gz. Be careful to submit only code, Makefile, README and not the binaries.

Procedures -

For additional announcements, monitor the course forum at piazza.com. The full address is https://piazza.com/uoa.gr/spring2019/k24/home. Attending the Piazza Forum is mandatory. - Your program should be written in C (or C++). In case you use C++ you cannot use the ready-made structures of the Standard Template Library (STL). In any case, your program should run on the Department's Linux workstations.

- Your code should consist of at least two (and preferably more) different files. Using separate compilation is mandatory and your code should have a Makefile.

- Make sure you follow good software engineering practices when implementing the exercise. Organization, readability, and presence of comments in the code are part of the score your.
- In the first two lessons, a hard-copy list is circulated in the classroom in which you should definitely give your name and your Unix user-id. This way we can know that you intend to submit this exercise and take the appropriate actions for the final submission of the exercise.

Other important notes -

Assignments are individual. -

Anyone who submits / presents code not written by him/her is zeroed in lesson.

- Although you are expected to discuss with friends and colleagues how you will attempt to resolve the problem, copying code (in any form) is not permitted. Anyone found involved in copying code simply gets a zero in the class. This applies to those involved regardless of who gave/received etc.

- Programming exercises can be given with a delay of a maximum of 3 days and with a penalty of 5% for every day of delay. Beyond these 3 days, exercises cannot be submitted.