Ονοματεπώνυμο: Παντελεήμων Μαλέκας
A.M: 1115201600268

Compilation command 1: make (with Makefile)
Compilation command 2: gcc -o bitcoin bitcoin.c Hash_Table.c Tree.c Transaction.c Wallet.c (without Makefile)

Run command: ./bitcoin -a bitCoinBalancesFile -t transactionsFile -v bitCoinValue -h1 senderHashtableNumOfEntries -h2 receiverHashtableNumOfEntries -b bucketSize

In order for the program to work the arguments must be given in that order.

When the program begins it reads the information from bitCoinBalancesFile with fscanf function. A singly linked list is used to store most of the information. Specifically, the users are placed in a wallet list, and the bitcoins (which are represented through tree_node structure) are placed in a bit list. Also each wallet has the same bit list struct as a data member to store his individual bitcoins.
If we stumble upon two wallets or two bitcoins with the same ID the program finishes with assert function.

After storing all the information from bitCoinBalancesFile the program reads and stores the information from transactionsFile, again with fscanf. After reading each member its validity is checked. If we stumble upon something which is not valid, (i.e. not enough money in sender's wallet) then we skip all the characters until \n to proceed to the next transaction. If everything is valid then we add the struct transaction to a similar linked list of transactions. Then the program proceeds to update the structs that participate. First we check how many bitcoins will be used. If one is necessary then we call functions for one bitcoin (example: Break_bitcoin_one, update_sender_one etc). If we need more than one, we call different functions (Break_bitcoin_list, update_sender_many etc). One temp tree_node is used in the first case and a temp bit_list in the second. After getting the bitcoin(s) we break the necessary nodes to left and right ones (if needed), then we update the bit_lists of sender and receiver. After this work, we create and proceed to the hash tables. Their implementation is as follows: A hash_table struct contains an array in which every cell contains the head of a linked list of buckets. Each bucket list node contains an array of struct bucket data(in other words,the bucket), a pointer to the next node, and information of whether the bucket is full or not. Bucket data struct contains what is placed at every cell of the bucket array. Specifically, the user's name and the head of a linked list of pointers to a struct transaction. After initializing and creating the hash_tables we use a hash function, which adds the ASCII characters of each user's ID and does % operation with the number of entries. If the key returned doesn't exist we create the first bucket_list node and we place him in the first spot of the bucket array, and we also create the first node of his transactions list. If the key returned does exist, first we search the the bucket list to find the user. If he exists, we add a transaction node at the end of his list. If he doesn't exist we add him

at the first available spot. If the bucket is full a new bucket list node is created and he is placed at the first spot.
After doing all the necessary work for the transactionsFile, we proceed to the command menu. The user may type any of the following commands:
(Arguments in brackets are optional)
requestTransaction sender_ID receiver_ID amount [date] [time]
requestTransactions sender_ID receiver_ID amount [date] [time];
requestTransactions InputFile.txt
findEarnings walletID [time1] [date1] [time1] [date2];
findPayments walletID [time1] [date1] [time1] [date2]
walletStatus walletID
bitCoinStatus bitCoinID
traceCoin bitCoinID
exit

Important note: For the commands to work properly please type the desired input then enter. No unnecessary spaces in between. For example please do not type: walletStatus richard \n

A few notes for each command:
requestTransaction reads the input from the user, and after checking the validity of each argument it updates the data structures as explained previously. For the IDs we generate a random number that is not already in the other transactions and we increment it for each transaction. If no date and time are given we get PC's local time.

requestTransactions reads multiple transactions. It is implemented similarly to the previous command. Please note the ; character must be typed at every transaction given. Also if one wishes to enter the loop the first transaction must be valid. Otherwise we return to the main menu. If the user enters the loop he may type as many transactions as he wishes. If he wishes to exit he simply types 0.

requestTransactions InputFile.txt reads data from an input file and does similar work as explained above. Please note the .txt must be typed along with InputFile. If someone types "requestTransactions InputFile" then it is not vaild.

findEarnings/findPayments search each hash table (receiver for earings, sender for payments). They add all the neccessary amount from the transactions of the given user and also print each transaction's information. If one wishes he may also give starting/ending times or starting/ending dates, or both. If only one of them is typed then the input is invalid.

walletStatus prints how much money the given user has in his wallet.

bitCoinStatus prints the the bitcoinID, how many transactions used it and its unspent value.

traceCoin prints the transactions where the given bitcoin was used.
exit command finishes the menu. After exiting we free the memory that was allocated.

Sources: The only source that I used was “Η γλώσσα C σε βάθος” in order to create the tree.