Test and Assessment - Print View

Final Exam (Homework)

Date: Mon Jun 21 17:08:22 2021 Maximum Points: 324

Question 1 - Build a hash function from compression function (10 Points) [ID: 872]

Consider the compression function $f : B^8 \to B^4$ given by

$f(b_0b_1b_2b_3b_4b_5b_6b_7) = e_0e_1e_2e_3$

where $e_i = b_{2i} \oplus b_{2i+1}$

Let further $x = 10111101010001010111$.

- Apply the Merkle-Damgaard procedure to construct a hash function $h : B^* \to B^4$ from f and calculate $h(x)$. Explain each step in this calculation.
- Give a collision of the hash function h.

Unlimited characters allowed, entered number of characters: **0**

Question 2 - Bloom filter - understanding (4 Points) [ID: 874]

Consider a Bloom filter after entering all elements of a set M. What is the output of the check(x) operation of the Bloom filter in the following cases?

1. Suppose that $x\in M.$
2. Suppose that $x\notin M.$

Unlimited characters allowed, entered number of characters: **0**

Question 3 - Bloom filter - using (20 Points) [ID: 876]

Let m=15 and consider the following hash functions

h1(x) = (17x+1) % 15

h2(x) = (11x+12) % 15

h3(x) = (13x+6) % 15

Enter the numbers 17, 42, 21 into the empty Bloom filter with these parameters.

Write down the resulting bitvector using the format BITVECTOR:000000000000000 (change 0 to 1 for bits that are set, the leftmost bit has index 0).

For each of the numbers 4, 5, 6, 7, 8, 9 determine whether the filter contains them! Write your answers in the form 4:YES or 4:NO (if 4 is/is not contained) etc up to 9:YES or 9:NO.

Separate answers by a single space, but do not use spaces inside an answer!

The order of the answers doesn't matter.

Score is granted based on the occurrence of the following keywords: - BITVECTOR:001111000110110 for 8 Points - 4:NO for 2 Points - 5:NO for 2 Points - 6:YES for 2 Points - 7:NO for 2 Points - 8:YES for 2 Points - 9:NO for 2 Points Scoring Mode: Automatic Scoring with Keywords on Finding ANY
72 characters allowed, entered number of characters: **0**

Question 4 - Key reconstruction from two fragments (10 Points) [ID: 878]

Alice has distributed fragments of her secret key using the linear interpolation scheme, where the key can be reconstructed from any two fragments. Her calculations are performed modulo 3947. Max obtained the following two fragments

- (5, 276)
- (250, 889)

Help Bob to calculate the secret and the random number used to hide the secret.

Write your answers in the format SECRET1111 and RANDOM1111 where you replace 1111 by the respective answer. Separate by spaces. Use exactly four digits. Fill in leading zeroes if necessary.

Score is granted based on the occurrence of the following keywords: - SECRET=2345 for 5 Points - RANDOM=1499 for 5 Points Scoring Mode: Automatic Scoring with Keywords on Finding ANY
25 characters allowed, entered number of characters: **0**

Question 5 - Mining Strategies (15 Points) [ID: 880]

1. Explain the concepts proof-of-work and proof-of-stake.
2. Discuss the advantages and disadvantages of POW and POS.
3. Explain the motivation behind the design of the SCRYPT hash function.

Unlimited characters allowed, entered number of characters: **0**

Question 6 - Contracts Ethereum vs Tezos (10 Points) [ID: 882]

A contract on Ethereum/Solidity can directly transfer money or invoke a method in another contract. In contrast, in contract on Tezos/Michelson cannot directly transfer money or invoke another contract.

Explain how a Michelson contract can initiate a transfer. Give some reasons why Tezos/Michelson avoids direct transfers.

Unlimited characters allowed, entered number of characters: **0**

Question 7 - Michelson Typing (10 Points) [ID: 884]

Each sequence of instructions in Michelson has a type that indicates which elements it expects on the stack and what kind of elements it leaves behind. Consider the following sequence of instructions

0: DUP;

1: CAR;

2: DIP(CDR);

3: CONCAT

4:

- Give valid stack typings in front of each instruction in the following sequence and after the sequence (that is, for each numbers such that the type of each instruction fits with your typings.

Hint: start with a valid stack typing at the end and push the information forward according to the type of the instruction. The numbers do not belong to the instruction sequence, but you may use them to refer to instructions in your solution.

Unlimited characters allowed, entered number of characters: **0**

Question 8 - Hashed Datastructures for Ethereum (20 Points) [ID: 886]

Research the internet to find out about the data structure which is used by the Ethereum blockchain to store its state (e.g., account balances, state of the contracts). Let's call this data structure ES. Some parts of ES are discussed in the lecture.

1. Give the official name of ES and explain those parts which are not discussed in the lecture.
2. Give an algorithm (in pseudocode) to insert data into an instance of ES.
3. How does Ethereum store maps in ES?

Unlimited characters allowed, entered number of characters: **0**

Question 9 - A lottery contract (15 Points) [ID: 888]

Consider a contract for a lottery on the Ethereum blockchain. Each participant chooses a secret number. To avoid publishing the number prematurely, they submit the hash of the secret along with their stake to the contract. Once all bids are in, participants are asked to reveal their secrets by submitting their secret number and having the contract check that it hashes to the previously submitted hash. Once all secrets are revealed, the contract calculates the winner from the “secret” numbers and transfers all collected stakes to the winner.

This outline of the contract has two problems. Identify the problems and explain how to change the contract to avoid the problems.

Unlimited characters allowed, entered number of characters: **0**

Question 10 - Solidity - Crowdfunding (60 Points) [ID: 890]

Write a contract for crowdfunding on Ethereum using the Solidity language. The contract will be created with a funding goal and will start in an open state where deposits are accepted. The contract can be closed as soon as the funding goal is met. When the contract is closed (by having the owner call the method **close**, but this must not happen before the funding goal is reached), all deposits are transferred to the owner of the contract and no further interaction with the contract will be possible. Only the owner can invoke **close**.

While the contract is open,

- everyone can use the **pay_in** method (as often as they want) to deposit funds in the contract;
- everyone can use the **withdraw** method to get their total deposit back.

Your contract may have further private methods beyond the required public methods **close**, **pay_in**, **withdraw**, and the constructor.

Upload your solution as a solidity file with extension .sol

Upload File

Choose File No file selected

Upload

Maximum upload size: 5000.0 MB

Allowed File Extensions: sol

(60 Points)

Question 11 - Michelson-coding (60 Points) [ID: 892]

- auction.tz (0 B)
- Tezos-Michelson-coding.pdf (0 B)

See attached file **Tezos-Michelson-coding.pdf** for the question

Upload File

Choose File No file selected

Upload

Maximum upload size: 5000.0 MB

(60 Points)

Question 12 - Examcoin (90 Points) [ID: 894]

- examcoin.pdf (0 B)
- examcoin.py (0 B)

See the attached file examcoin.pdf for the question.

The attached examcoin.py contains a code template to get you started.

Upload File

Choose File No file selected

Upload

Maximum upload size: 5000.0 MB

(90 Points)

Given a smart contract called *auction* as

```
parameter (pair mutez address); # the participant's bid amount and address
storage (pair
            (pair int address) # the number of accepted bids made by
                               participants and the contract owner's address
            (pair mutez address) # the current highest bidder's bid amount and
                                address
         );
code {# (pair parameter storage) : []
       # pair (pair @parameter mutez address) (pair @storage (pair int address) (pair mutez
       address)) : []
       # make sure that the participant has contributed at the minimal price 2 tez
       PUSH mutez 2000000;
       AMOUNT; # get the amount bid that is transferred along with this contract call
       IFCMPGE {} { PUSH string "You did not provide enough tez (the minimal
       price 2 tez)."; FAILWITH; };
       # check that the number of accepted bids has not been exceeded 10
       UNPAIR;
       # pair mutez address : pair (pair int address) (pair mutez address) : []
       SWAP;
       # pair (pair int address) (pair mutez address) : pair mutez address : []
       DUP;
       # pair (pair int address) (pair mutez address) : pair (pair int address) (pair mutez
       address) : pair mutez address : []
       CAR;
       # pair int address : pair (pair int address) (pair mutez address) : pair mutez
       address : []
       CAR;
       # int : pair (pair int address) (pair mutez address) : pair mutez address : []
       DIP { PUSH int 10; };
       # int : int : pair (pair int address) (pair mutez address) : pair mutez address : []
       IFCMPLT {
                  # check if the participant's bid is higher than the current  highest bid
                    SWAP;
                    # pair mutez address : pair (pair int address) (pair mutez address): []
                    DIP { DUP; };
                    # pair mutez address : pair (pair int address) (pair mutez
                    address) : pair (pair int address) (pair mutez address) : []
                    DUP;
                    # pair mutez address : pair mutez address : pair (pair int
                    address) (pair mutez address) : pair (pair int address) (pair mutez
                    address) : []
                    DIP { SWAP; };
                    # pair mutez address : pair (pair int address) (pair mutez
                    address) : pair mutez address : pair (pair int address) (pair mutez
                    address) : []
                    DIP { CDR; };
                    # pair mutez address : pair mutez address : pair mutez address    :
                    pair (pair int address) (pair mutez address) : []
                    CAR;
```

```
# mutez : pair mutez address : pair mutez address : pair (pair int
address) (pair mutez address) : []
DIP { CAR; };
# mutez : mutez : pair mutez address : pair (pair int address) (pair
mutez address) : []
IFCMPGT {
            # If the participant' bid is higher than the
            current highest bid, the contract returns the
            previous bid to its owner and updates the store
            with the new current highest bid amount and the
            bidder’s address

            SWAP;
            # pair (pair int address) (pair mutez address) :
            pair mutez address : []
            UNPAIR;
            # pair int address : pair mutez address : pair
            mutez address : []
            SWAP;
            # pair mutez address : pair int address : pair
            mutez address : []
            UNPAIR;
            # mutez : address : pair int address : pair mutez
            address : []
            SWAP;
            # address : mutez : pair int address : pair mutez
            address : []
            CONTRACT unit;
            # convert the current highest bidder's address to
            a contract
            IF_SOME {} { FAILWITH; };
            # contract unit : mutez : pair int address : pair mutez
                  address : []
            SWAP;
            # mutez : contract unit : pair int address : pair mutez
            address : []
            DUP;
            # mutez : mutez : contract : pair int address : pair
            mutez address : []
            PUSH mutez 0;
            # mutez : mutez : mutez : contract unit : pair int
            address : pair mutez address : []
            IFCMPLT {
                  # mutez : contract unit : pair int address : pair
                        mutez address : []
                  UNIT; # match the contract type
                  # unit : mutez : contract unit : pair int address :
                  pair mutez address : []
                  TRANSFER_TOKENS; # transfer the money to
                  the current highest bidder
                  # operation : pair int address : pair mutez
                  address : []
                  PUSH int 1;
```

```
                # int : operation : pair int address : pair
                mutez address : []
                SWAP;
                # operation : int : pair int address : pair
                mutez address : []
                DIP 2 { UNPAIR; };
                # operation : int : int : address : pair
                mutez address : []
                DIP { ADD; };
                # operation : int : address : pair mutez
                address : []
                DIP { PAIR; };
                # operation : pair int address : pair mutez
                address : []
                DIP { PAIR; };
                # operation : pair (pair int address) (pair
                mutez address) : []
                NIL operation; SWAP; CONS; PAIR;
                # pair (list operation) (pair (pair int
                address) (pair mutez address)) : []
        }
        {
                # mutez : contract unit : pair int address : pair
                mutez address : []
                DROP;
                # contract unit : pair int address : pair mutez
                address : []
                DROP;
                # pair int address : pair mutez address : []
                UNPAIR;
                # int : address : pair mutez address : []
                PUSH int 1;
                # int : int : address : pair mutez address : []
                ADD;
                # int : address : pair mutez address : []
                PAIR;
                # pair int address : pair mutez address : []
                PAIR;
                # pair (pair int address) (pair mutez
                address) : []
                NIL operation; PAIR;
                # pair (list operation) (pair (pair int
                address) (pair mutez address)) : []
        }
}
{       # If the participant' bid is lower than the current
        highest bid, the contract rejects the bid.
        # pair mutez address : pair (pair int address)
        (pair mutez address) : []
        PUSH string "your bid is lower than or equal to
        the current highest bid."; FAILWITH;
```

```
                    };
                }
                {
                    # if the number of accepted bids exceeds 10
                    PUSH string "The auction is closed since the number of
                    accepted bids exceeds 10."; FAILWITH;
                };
            };
```

- black text is Michelson instructions.

- blue text is comments.

Intuitively, the auction smart contract works according to the following strategy

- The contract manager originates the contract and provides the owner's address.

- The input parameter contains a pair of the bid amount and the bidder's address

```
(pair mutez address)
```

- The storage of the contract contains a pair of

(1) a pair of the number of bids made by participants and the contract owner's address

(2) a pair of the current highest bid amount and the bidder's address

```
(pair
      (pair int address)
      (pair mutez address)
)
```

- The contract will be closed when there are 10 accepted bids (an accepted bid is higher than the previous highest bid).

- A bid is placed by sending money (a bid) to the contract: if the amount of the bid is less or equal than the current highest bid, the contract rejects the bid and this is not considered as an accepted bid. Otherwise, the contract returns the previous bid to its owner and updates the store with the new current highest bid amount and the bidder's address.

1. Given the initial stack as

S :: Pair (Pair 9 "tz1VmefgyE1YPcAa8VurxihsUJXzLUX6HY8Z")
(Pair (Pair 3 "tz1XbA8xcxkPAkB7CwyNq4fnbQdCXunKXHsb")
(Pair 7 "tz1XJu9hk9aYZGqrAWZDsDxMHRoUQ1NwC5CL")) : [ ]

Give the content of the stack after each instruction.

2. The current version does not validate whether the amount that is transferred along with a smart contract call is the same as what is claimed in the input parameter.

Revise the code of the smart contract such that it validates whether these two amounts are the same.

Hint.

- the input parameter is a pair **`(pair mutez address)`**

- the instruction **`AMOUNT`** pushes the amount that is sent along with the contract call.

3. In the current version of the auction contract, when there are 10 accepted bids, the auction is closed as follows:

```
{
      # if the number of accepted bids exceeds 10
      PUSH string "The auction is closed since the number of accepted
      bids exceeds 10."; FAILWITH;
};
```

Revise the code such that it will transfer the current highest bid to the contract owner. (Your code does not have to avoid multiple transfers to the owner. Explain why.)

```
{
      # if the number of accepted bids exceeds 10
      # write the code here
      ...
};
```

4. Each bidder in the auction contract should either win the auction or get the bid back. Discuss whether this statement is true for the given code and propose a solution if it is not.

Implement the *Examcoin* system that is a simple version of Bitcoin and works according to the following strategies:

1. A block contains the following information

prev_hash	The hash of the previous block
merkle_root	The root of the Merkle-Tree of this block's transactions
timestamp	The approximate creation time of this block
nonce	The nonce, a counter used for the proof-of-work algorithm
difficulty	The proof-of-work algorithm difficulty target for this block
prev	The pointer points to the previous block
data	The set of transactions

2. There are two kinds of transactions

A. CoinsBase

When a miner mines a new block they will be awarded 20 coins. Here is an example transaction in json format (Python dictionary):

{"hash": "707e3e923d4e0b6c3c54ae6880659a9bb371cb7dd556",

"type": "CoinsBase",

"coins_created": [{"num": 0, "value": 20, "recipient": "[Miner]"}],

"Timestamp": "2020-08-24 14:24:46.986531"}

where [Miner] is the miner's public key (or the miner's alias).

B. PayCoins

A user can transfer his/her coins to another. The transaction includes a list of unspent coins that are owned by the user. The transaction creates new coins for the new user and may transfer back the rest of the coins to the old user. For example, Alice owns 5.5 unspent coins and she wants to transfer 4 coins to Bob. Alice's unspent coins that are used in this transaction are listed in the "coins_consumed" field.

{"hash": "242874663bcf7485099cb993ceb46e0b42594e20319e56",

"type": "PayCoins",

"coins_consumed": [

{"hash": "f1531ef8c83c9d37d68f2505b2a4809d416cfe1b101c27", "num": 0},

{"hash": "bbf6b700cf24734e2874663bcf52f47485099cb993ceb4", "num": 1}],

"coins_created": [

{"num": 0, "value": 4, "recipient": "[Bob]"},

{"num": 1, "value": 1.2, "recipient": "[Alice]"}],

"transaction fee" : 0.3

"signatures" : "signature of [Alice]",

"Timestamp": "2020-08-24 15:54:46.983531"}

2. When a user creates a new transaction, this transaction is stored in the pending pool that contains all pending transactions before they may be included to a block.

3. Miners mine a new block with a proof of work algorithm as in Bitcoin.

- The time to mine a new block is based on the difficulty. Assuming that the difficulty starts with 10 and it increases 20 % after 10 blocks created.

difficulty = 10

maxNonce = 2 ** 32

target = 2 ** (256 – difficulty)

- A miner collects transactions in the pending pool. Maximum 10 transactions would be included in one block. Lets us assume that there are three ways to collect the transactions for the block.

+ Miners collect transactions based on their transaction fees. Transactions with high transaction fees get selected first.

+ Miners collect 10 transactions based on the timestamp. Transactions with early timestamp get selected first.

+ Miners collect 10 transactions randomly.

- Miners build a Merkle tree of these selected transactions and include the Merkle tree root in the block.

- Before adding the new block to the block chain, a miner should validate the chain. The new block is supposed to be appended to the current head of the chain. However, if the head is invalid the miner is allowed to skip it.

- A miner receives 20 coins as reward for each new valid block created.

- A transaction is valid if

For CoinsBase:

+ the number of coins created is 20.

For PayCoins:

+ The coins spent plus the transaction fee is less or equal to the number consumed coins.

+ The consumed coins are unspent (no double spending).

+ The signature is valid.

4. The system should provide functions that allow users to validate transactions, blocks, and chains.

5. The system should provide a function call ***getBalance*** that allows users to check the number of coins that they own.

6. The system should provide a transfer function that transfers n coins from [A] to [B] with fee m ***transfer(n, m, [A], [B])***

where

- ***[A]*** and ***[B]*** are public keys (or alias)

- ***n*** is the amount of coins that ***[A]*** wants to transfer to ***[B]***

- ***m*** is the transaction fee

The results of the transfer function could be

- a proposal for a transaction that performs the transfer

{"hash": "...",

"type": "PayCoins",

"coins_consumed": [

{"hash": "...", "num": ...},

{"hash": "...", "num": ...},

…],

"coins_created": [

{"num": 0, "value": n, "recipient": "[B]"},

{"num": 1, "value": n', "recipient": "[A]"}],

"transaction fee" : m,

"signatures" : "signature of [A]",

"Timestamp": "…"}

This transfer function should search for unspent coins of [A] on the blockchain. They could be from one transaction or more such that the total amount of them is equal or greater than n + m

- Fail if the balance is too low.

Hint.
We suggest the following structure for Python code that implements the *examcoin* system. This way we can assess each step and give marks for partial solutions. If you use a different approach, you should structure into steps that can be assessed individually (if you want marks for a partial solution). In any case, meaningful comments are part of the solution.

1. The whole program could be structured into one main program *examcoin* and two subprograms as follows.

- The *crypto_key* program that implements all functions to generate private/public keys, sign a document, and verify a signature.

- The *merkletree* program that implements all functions to construct a Merkle tree.

2. The main *examcoin* program should contain three classes Transaction, Block, and Examcoin as follows.

- black text is Python code.

- blue text is comments.

```
class Transaction:
	# the attribute _data is in a json format
	_data = None

	# 1. the constructor function
	def __init__(self, data):
		self._data = data
		self._data["Timestamp"] = str(datetime.datetime.now())
		self._data["hash"] = self.digest()

	# 2. there should be two different digest functions for hashing a transaction.
	# 2.1. this function returns a hash to be signed on. It is the hash of the fields
	"type", "coins_consumed", "coins_created" and "transaction fee" for "PayCoins" and
	"type" and "coins_created" for "CoinsBase"
	def digest_sign(self):

		# write your code here

	# 2.2. this function returns the hash of all fields
	def digest(self):

		# write your code here

	# 3. this function signs on a transaction by a given private key (or an alias that
	names to a private key file, which may require a password to open). Note that,
	after signing, the signature is appended to the field "signatures" of the
	attribute _data.
	def sign(self, private_key):
	# or def sign(self, alias, password):
```

```
            # write your code here

        # 4. this function verifies whether the signature of a transaction, which is in
        the "signatures" field is valid with a given public key (or an alias that names
        for a public key file)
        def verify_sig(self, public_key):
        # or def verify_sig(self, alias):

            # write your code here

        # 5. this function returns the total amount of coins created in a transaction,
        def get_sum_coin_created(self):

            # write your code here

        # 6. this function returns the value of a coin created
        def get_value(self, num):

            # write your code here

        # 7. there may be other functions.

        # write your code for other functions here


class Block:
        _prev_hash = bytearray(256)
        _merkle_root = bytearray(256)
        _timestamp = None
        _nonce = 0
        _difficulty = 0
        _prev = None
        _data = []

        # 1. the constructor function
        def __init__ (self, data, nonce, difficulty, prev = None):
            self._prev_hash = prev.digest() if prev is not None else
                              bytearray(256)
            self._merkle_root = construct_Merkletree(data)
            self._nonce = nonce
            self._difficulty = difficulty
            self._prev = prev
            self._data = data

         # 2. there should be two digest functions
        # 2.1. this function returns the hash of a block for the mining purpose. This hash
        is used to compare with the target in the proof of work algorithm. The data to be
        hashed includes _prev_hash, _nonce and _data.
        def mining_digest(self):

            # write your code here

        # 2.2. this function returns the hash of a block. The data to be hashed includes
        all the attributes
        def digest(self):

            # write your code here

        # 3. this function returns the value of a coin created
        def get_value_coin (self, trans_hash, num):

            # write the code here
```

```
	# 4. there may be other functions

	# write your code for other functions here

class examcoin:
	_head = None
	_root_hash = bytearray(256)
	_diff = 10
	_maxNonce = 2 ** 32
	_target = 2 ** (256 - _diff)
	_pending_pool = {}
	_level = 0

	# 1. the constructor function.
	def __init__(self):
		self._head = Block([], 0, 0, None)
		self._root_hash = self._head.digest()

	# 2. this function adds a new transaction to the pending pool.
	def add_transaction(self, trans):

		# write your code here.

	# 3. this function adds a new block to the chain. Note that this block need to
	contain a transaction that awards 20 coins to the minier. When a new block is
	added, the attribute _level should be increased by 1 and all the transactions that
	are included in this block should be removed from the pending pool.
	def __add_block(self, block, minier):
		json = {"hash": 0,
			"type": "CoinsBase",
			"coins_created": [
				{"num": 0, "value": 20, "recipient": minier}],
			"Timestamp": ""}
		award = Transaction(json)
		block._data.append(award)

		# write your code here.

	# 3.1. this function removes all selected transactions from the pending pool.
	def __filtering_pending_pool(self, selected_trans):

		# write your code here.

	# 4. this function checks whether a given transaction is valid.
	def verify_trans(self, trans):

		# write your code here.

	# there are some functions that are useful to validate a transaction.
	# 4. 1. this function returns the owner of a coin,
	def get_coin_owner(self, trans_hash, num):

		# write your code here.

	# 4.2. this functions checks whether the coins consumed in a transaction are
	double spending.
	def check_double_spending(self, coins_consumed):

		# write the code here.

	# 5. this function validates whether a given block is valid.
	def verify_block(self, block):
		# check the validation of the mining, the merkel tree root and all the
		transactions.
```

```
        # write your code here.

    # 6. this functions checks whether the chain is valid.
    def verify_chain(self):

        # write your code here.

    # 7. this function collects all valid transactions from the pending pool.
    def collect_valid_trans(self, pending_pool):

        # write your code here.

    # 8. given an array of valid transactions:
    # 8.1. this function returns an array of maximum 10 valid transactions sorted by
    transaction fee.
    def sort_trans_by_fee(self, valid_trans):

        # write your code here.

    # 8.2. this function returns an array of maximum 10 valid transactions sorted by
    timestamp.
    def sort_trans_by_timstamp(self, valid_trans):

        # write your code here.

    # 8.3. this function returns an array of maximum 10 valid transactions randomly.
    def sort_trans_by_random(self, valid_trans):

        # write your code here.

     # 9. this function mines a new block. The arguments are the public key (or
    alias), the ming method that could be "fee",  "timestamp" or "random" and the
    block that the new block want to append after (as default, it is the current head
    of the chain).
    def mine(self, alias, method = "fee", head = None):
    # the mining strategy is the proof of work algorithm as in Bitcoin. The nonce,
    difficulty (target) are used for mining. Note that the difficulty increases 20%
    after each cycle of 10 blocks created. The attribute _level could be used for this
    purpose.

        # write your code here.

    # 10. this function returns the balance of an account (public key or alias). The
    balance of an account is the value of its unspent coins.
    def get_balance(self, public_key):
        # or def get_balance(self, alias):

        # write your code here.

    # 11. this function performs a transfer that transfers the amount "amount" of
    coins from an account "from_addr" (private_key or alias) to another account
    "to_adrr" (public key or alias) with the transaction fee "fee".
    def transfer(self, amount, fee, from_addr, to_addr):
        # the function need to find the unspent coins of "from_addr" account, which
        could be one coin or more such that the sum of their values are equal or
        greater than "amount" plus "fee".
        # the transfer is fail if the balance of the account "from_addr" is too low.
        # if the transfer is successful, the transaction is added to the pending
        pool.

        # write your code here.

    # 12. there may be other functions.

    # write your code for other functions here.
```