Contents

1 README

```
daniel023, alonnetser
===============================================================================
Daniel Afrimi, ID 203865837, daniel.afrimi@mail.huji.ac.il
Alon Netser, ID 311602536, alon.netser@mail.huji.ac.il
===============================================================================

                                  Project 2
                           -----------------------


Submitted Files
---------------
README - This file.
ex2.py - Implementation of the second exercise.

Remarks
-------
In this project we implemented prototype of a secure payment system,
while in the first exercise we had single trusted authority (the Bank).
In this project we replaced the Bank with nodes that
running according to the longest-chain protocol.
```

2 ex2.py

```python
import sys
from typing import List, Optional, NewType, Set, Dict, Tuple
import ecdsa  # type: ignore
import hashlib
import secrets
from collections import deque


PublicKey = NewType('PublicKey', bytes)
Signature = NewType('Signature', bytes)
BlockHash = NewType('BlockHash', bytes)  # This will be the hash of a block
TxID = NewType("TxID", bytes)  # this will be a hash of a transaction

GENESIS_BLOCK_PREV = BlockHash(b"Genesis")  # these are the bytes written as the prev_block_hash of the 1st block.

BLOCK_SIZE = 10  # The maximal size of a block. Larger blocks are illegal.


def hash_function(input_bytes: bytes) -> bytes:
    """
    This function hashes the given input bytes, using the SHA256 hash-function.
    It mainly helps avoid writing '.digest()' everywhere...
    :param input_bytes: The input bytes to calculate the hash from.
    :return: The hash of the input bytes.
    """
    return hashlib.sha256(input_bytes).digest()


def get_transactions_hash(transactions: List['Transaction']) -> bytes:
    """
    Get the hash of all of the transactions in the block.
    The hash is the merkle-tree root.
    We handle cases of odd number of children by duplicating the transaction, as described in:
    https://bitcoin.stackexchange.com/questions/46767/merkle-tree-structure-for-9-transactions
    """
    # If there are no transactions in the block, return the hash "nothing" (i.e. empty bytes).
    if len(transactions) == 0:
        return hash_function(bytes())

    # If there is only one transaction in the block, the hash of the transactions is just its
    # TxID (which is already the hash of the transaction).
    if len(transactions) == 1:
        return bytes(transactions[0].get_txid())

    # Now we know that there are at least 2 transactions in the block, so let's create the merkle-tree.

    # The will serve as a queue (FIFO data-structure) to process the tree from to bottom to the top.
    curr_queue = deque([bytes(tx.get_txid()) for tx in transactions])

    # As long as the queue contains at least 2 elements, hash every pair and insert
    # it to the right hand-side of the queue.
    while len(curr_queue) >= 2:
        # The amount of times the for-loop will execute is n // 2, and if
        # n is odd then the last transaction is repeated to form a "pair" of transactions.
        n: int = len(curr_queue)
        for i in range(0, n, 2):
            left_txid: bytes = curr_queue.popleft()
            right_txid: bytes = curr_queue.popleft() if i + 1 < n else left_txid

```

```python
            curr_hash: bytes = hash_function(left_txid + right_txid)
            curr_queue.append(curr_hash)

    # Now we know that the queue has only one element, so it's the root of the merkle tree.
    merkle_root: bytes = curr_queue.pop()

    return merkle_root


class Transaction:
    """
    Represents a transaction that moves a single coin.
    A transaction with no source creates money. It will only be created by the miner of a block.
    Instead of a signature, it should have 48 random bytes.
    """

    def __init__(self, output: PublicKey, tx_input: Optional[TxID], signature: Signature) -> None:
        """
        Initialize the transaction object with the given output, input and signature.
        :param output: The output of the transaction, which is the address (a.k.a. public_key)
                       of the recipient.
        :param tx_input: The input of the transaction, which is the transaction ID to use
                        (its output is the sender public_key).
        :param signature: The signature of the sender, which uses the private_key associated
                          with the input-transaction's output public_key.
                          The message that is being signed is the concatenation of the
                          recipient's public_key and the input transaction ID.
        """
        self.output: PublicKey = output  # DO NOT change these field names.
        self.input: Optional[TxID] = tx_input  # DO NOT change these field names.
        self.signature: Signature = signature  # DO NOT change these field names.

    def get_txid(self) -> TxID:
        """
        Returns the identifier of this transaction. This is the sha256 of the transaction contents.
        :return: The ID of this transaction, which is the hash of the concatenation of
                 the output, input and the signature.
                 Note that if the input is None then it's regarded as empty bytes in the concatenation.
        """
        input_bytes: bytes = bytes() if self.input is None else self.input
        tx_content: bytes = self.output + input_bytes + self.signature
        return TxID(hash_function(tx_content))

    def __hash__(self):
        """
        This function is implemented to enable storing the Transaction object in hashable containers (such as a set).
        Note that it's crucial for the test_longer_chain_overtake because there is
        assert set(bob.get_utxo()) == set(alice.get_utxo())
        So in order to create a set of transactions it must be hashable.

        :return: The hash of this Transaction object.
        """
        return int.from_bytes(self.get_txid(), byteorder=sys.byteorder)

    def __eq__(self, other: 'Transaction'):
        """
        This function is implemented to enable storing the Transaction object in hashable containers (such as a set),
        as well as to allow doing things like 'tx in tx_list' and it'll be true if a transaction in 'tx_list' has the
        same TxID.
        :param other: Another Transaction to check equality to.
        :return: True if and only if the other Transaction has the same TxID.
        """
        return self.get_txid() == other.get_txid()


class Block:
    """
    This class represents a block.
```

```python
    """
    def __init__(self,
                 previous_block_hash: BlockHash,
                 transactions: Optional[List[Transaction]] = None) -> None:
        """
        Initialize the Block object with the given transactions and previous block-hash.
        :param previous_block_hash: The block-hash of the previous block in the blockchain.
        :param transactions: The list of transactions to insert into this block.
                             If not given - the default is an empty-list.
        """
        # This is done to avoid using mutable values as a default argument (i.e. transactions=list()).
        if transactions is None:
            transactions: List[Transaction] = list()

        self.previous_block_hash: BlockHash = previous_block_hash
        self.transactions: List[Transaction] = transactions

    def get_block_hash(self) -> BlockHash:
        """
        Gets the hash of this block, which is the hash of the concatenation of the previous block-hash
        and the transactions' hash (i.e. the merkle root).
        :return: The hash of this block.
        """
        return BlockHash(hash_function(self.previous_block_hash + get_transactions_hash(self.transactions)))

    def get_transactions(self) -> List[Transaction]:
        """
        :return: The list of transactions in this block.
        """
        return self.transactions

    def get_prev_block_hash(self) -> BlockHash:
        """
        :return: The hash of the previous block
        """
        return self.previous_block_hash


class Node:
    def __init__(self) -> None:
        """
        Creates a new node with an empty mempool and no connections to others.
        Blocks mined by this nodes will reward the miner with a single new coin,
        created out of thin air and associated with the mining reward address.
        """
        self.blockchain: List[Block] = list()
        self.mempool: List[Transaction] = list()

        # This is the list of unspent transactions (to validate that new transactions
        # that are entering the MemPool use an unspent input).
        self.utxo: List[Transaction] = list()

        # This is used in order to get the block given the BlockHash in O(1),
        # instead of iterating the blockchain which takes O(len(blockchain)).
        self.block_hash_to_index: Dict[BlockHash, int] = dict()

        # This is used in order to get the block given the BlockHash in O(1),
        # instead of iterating the blockchain which takes O(len(blockchain)).
        self.txid_to_blockhash: Dict[TxID, BlockHash] = dict()

        self.private_key: ecdsa.SigningKey = ecdsa.SigningKey.generate()
        self.public_key: PublicKey = PublicKey(self.private_key.get_verifying_key().to_der())

        # These are the TxIDs of the transactions in the blockchain that their output is this Node's public_key.
        self.coins: List[TxID] = list()

        # These are the TxIDs of the transactions in the blockchain that their output is this Node's public_key.
        # BUT - this Node didn't use them already in creating new transactions
```

```python
        # (that maybe didn't made it into the blockchain yet).
        self.unspent_coins: List[TxID] = list()

        # This is the list of all the connections this Node has.
        self.connections: Set[Node] = set()

    def get_tx_index_in_utxo(self, transaction: Transaction) -> int:
        """
        Get the index of the transaction in the UTxO that its TxID matches the TxID of the given transaction.
        :param transaction: The transaction to search
        :return: The index of the transaction in the UTxO that its TxID matches the TxID of the given transaction.
                 In case the transaction was not found, return -1.
        """
        for i, tx in enumerate(self.utxo):
            if tx.get_txid() == transaction.input:
                return i

        return -1

    def connect(self, other: 'Node') -> None:
        """
        Connects this node to another node for block and transaction updates.
        Connections are bi-directional, so the other node is connected to this one as well.
        Raises an exception if asked to connect to itself.
        The connection itself does not trigger updates about the mempool,
        but nodes instantly notify of their latest block to each other.
        """
        # Check if the given Node is equal to this Node. Equality is determined by the manually specified
        # __eq__ function, which state that two Node are equal if they have the same public-key.
        if self == other:
            raise ValueError("Can not add the Node as a connection of itself.")

        if other not in self.connections:
            # Establish the mutual connection between the two nodes.
            self.connections.add(other)

            # Upon connection, nodes notify each other about the tip of their blockchain.
            # Note that MemPool transactions are not shared upon connection.
            self.notify_of_block(other.get_latest_hash(), other)

            # The connection is mutual.
            other.connect(self)

    def disconnect_from(self, other: 'Node') -> None:
        """
        Disconnects this node from the other node. If the two were not connected, then nothing happens.
        """
        if other in self.connections:
            self.connections.remove(other)
            other.disconnect_from(self)

    def get_connections(self) -> Set['Node']:
        """
        Returns a set of the connections of this node.
        """
        return self.connections

    def notify_latest_block_to_all_connections(self) -> None:
        """
        Notify all connections of this node regarding the latest block in the blockchain.
        """
        for node in self.connections:
            node.notify_of_block(self.get_latest_hash(), self)

    def notify_transaction_to_all_connections(self, transaction: Transaction) -> None:
        """
        Notify all connections of this node regarding the given transaction (effectively adding it to their MemPool).

```

```python
        :param transaction: The transaction to notify about.
        """
        for node in self.connections:
            node.add_transaction_to_mempool(transaction)

    def verify_transaction_validity(self, transaction: Transaction, verify_with_mempool=True) -> bool:
        """
        Verify the validity of the transaction.
        It will return False iff any of the following conditions hold:
        (i) The transaction is invalid (the signature fails).
        (ii) The source doesn't have the coin that he tries to spend.
        (iii) There is contradicting tx in the mempool.

        :param transaction: The transaction to check validity.
        :param verify_with_mempool: Whether to verify that there is no contradicting tx in the MemPool or not.
                                    Will be true when adding a tx to our MemPool, and will be False when validating
                                    the txs of a new given chain (since their txs are not in our MemPool).
        :return: False iff any of the above conditions hold.
        """
        input_tx_index_in_utxo: int = self.get_tx_index_in_utxo(transaction)
        if input_tx_index_in_utxo == -1:
            return False

        input_transaction: Transaction = self.utxo[input_tx_index_in_utxo]
        public_key: PublicKey = input_transaction.output

        # Verify that the transaction is valid, using the public-key as a verifying key and the
        # (transaction's input + transaction's output) as the data that was signed.
        try:
            ecdsa.VerifyingKey.from_der(public_key).verify(signature=transaction.signature,
                                                           data=transaction.output + transaction.input)
        except ecdsa.BadSignatureError:
            return False

        if verify_with_mempool:
            # Verify that there is no contradicting transaction in the MemPool.
            # This means a transaction that uses the input TxID (disallow double-spending).
            if transaction.input in [tx.input for tx in self.mempool]:
                return False

        return True

    def add_transaction_to_mempool(self, transaction: Transaction, notify_connections: bool = True) -> bool:
        """
        This function inserts the given transaction to the mempool.
        It is used by a Node's connections to inform it of a new transaction.
        It will return False iff any of the following conditions hold:
        (i) The transaction is invalid (the signature fails).
        (ii) The source doesn't have the coin that he tries to spend.
        (iii) There is contradicting tx in the mempool.

        :param transaction: The transaction to add to the MemPool.
        :param notify_connections: Should we notify the connections regarding this new transaction.
                                   Will be False when the transaction being added is due to ReOrg
                                   (i.e. transactions in the removed chain that are still valid and
                                   can enter the blockchain).
        :return: True if it was added, False otherwise (because it was invalid).
        """
        transaction_is_valid: bool = self.verify_transaction_validity(transaction)

        if transaction_is_valid:
            self.mempool.append(transaction)

            if notify_connections:
                self.notify_transaction_to_all_connections(transaction)

        return transaction_is_valid

```

```python
    def build_alternative_chain(self, block_hash: BlockHash, sender: 'Node') -> List[Block]:
        """
        Build an alternative chain of blocks, starting from the given block_hash and going backwards until reaching
        a block in the node's blockchain (it might the the GENESIS block, meaning that the alternative chain
        replaces the whole blockchain of this node).
        :param block_hash: The block last block of the alternative blockchain.
        :param sender: The sender of the block_hash, will (possibly) request blocks from him.
        :return: The alternative list of blocks.
                 If the given block is already in the blockchain, an empty list is returned.
        """
        new_chain: List[Block] = list()

        while (block_hash not in self.block_hash_to_index) and (block_hash != GENESIS_BLOCK_PREV):

            try:
                block: Block = sender.get_block(block_hash)
            except ValueError:
                return list()

            # It is possible that a "bad" node will return a block with a different hash than requested.
            if block.get_block_hash() != block_hash:
                return list()

            new_chain.append(block)
            block_hash: BlockHash = block.get_prev_block_hash()

        # Reverse the new chain, because we appended the previous block to the right.
        # We reverse at the end and not insert to the left, because appending to the right of a list is O(1)
        # while inserting to the left is O(n).
        new_chain.reverse()

        return new_chain

    def get_transaction(self, txid: TxID) -> Optional[Transaction]:
        """
        Get a transaction given its TxID.
        This is done in O(1) since we save a mapping between TxID and the block-hash
        which holds this transaction in the blockchain.
        Iterating over the transactions in the block is also O(1) because the block is bounded in size
        (in this exercise maximum 10 transactions, but also in real life it's bounded,
        as opposed to the blockchain that is "unbounded" and it's indeed very long).
        :param txid: The TxID of a transaction in the blockchain to get.
        :return: The transaction.
                 returns None if it is not in the blockchain (should no happen, there is even an assert).
        """
        block_hash: BlockHash = self.txid_to_blockhash[txid]
        block: Block = self.blockchain[self.block_hash_to_index[block_hash]]
        for tx in block.get_transactions():
            if tx.get_txid() == txid:
                return tx

    def get_relevant_transactions_from_removed_chain(self, removed_chain: List[Block]) -> Tuple[List[TxID],
                                                                                                List[Transaction]]:
        """
        Get the relevant transactions from the removed chain, which are the input-transactions of transactions
        in the removed chain, that exists in the original blockchain (and not in the removed chain).
        It also returns the TxIDs of all of the removed transactions in the given removed chain (no exceptions).

        :param removed_chain: The removed chain of blocks.
        :return: removed_txids and input_transactions_now_unspent
        """
        removed_transactions: List[Transaction] = [tx for block in removed_chain for tx in block.get_transactions()]
        removed_txids: List[TxID] = [tx.get_txid() for tx in removed_transactions]

        # Keep only the removed transactions with input in the blockchain (and not in the removed blocks).
        # These transactions' inputs should be added to the UTxO, since they now are un-spent.
        removed_transactions_with_input_in_blockchain: List[Transaction] = [tx for tx in removed_transactions
                                                                            if tx.input not in removed_txids
```

```python
                                                                        and tx.input is not None]
        # Input transactions the are now unspent, because they were used in transactions in the removed blocks.
        input_transactions_now_unspent: List[Transaction] = [self.get_transaction(tx.input)
                                                             for tx in removed_transactions_with_input_in_blockchain]

        return removed_txids, input_transactions_now_unspent

    def update_utxo(self, removed_chain: List[Block]):
        """
        Update the UTxO, according to the removed transactions.
        Some transactions are now un-spent (because we removed a transaction that used
        some input transaction, so the input transaction is now un-spent).
        Some transactions were un-spent and now does not exists so they need to be removed.

        :param removed_chain: The list of all transactions that were removed from the blockchain.
        """
        removed_txids, input_transactions_now_unspent = self.get_relevant_transactions_from_removed_chain(removed_chain)

        # Extend the UTxO with the input TxID of transactions that were removed from the blockchain
        # and that the input-transaction was not in this removed transactions list.
        # Now these transactions are un-spent.
        self.utxo.extend(input_transactions_now_unspent)

        # Remove the transactions in the UTxO that were removed from the blockchain.
        self.utxo: List[Transaction] = [tx for tx in self.utxo if tx.get_txid() not in removed_txids]

    def update_coins_according_to_removed_chain(self, removed_chain: List[Block]):
        """
        Update the UTxO and the coins assigned to this node, according to the removed transactions.
        Some transactions are now un-spent (because we removed a transaction that used
        some input transaction, so the input transaction is now un-spent).
        Some transactions were un-spent and now does not exists so they need to be removed.

        :param removed_chain: The list of all transactions that were removed from the blockchain.
        """
        removed_txids, input_transactions_now_unspent = self.get_relevant_transactions_from_removed_chain(removed_chain)

        # coins_to_add are the input-transactions of transactions in the removed blocks
        # (as long as the input transaction is in the blockchain, and not in the removed blocks)
        # and the output of this input transaction is the current node.
        # This means coins that the node used but now since the blockchain is changing, he can use them again.
        coins_to_add: List[TxID] = [tx.get_txid() for tx in input_transactions_now_unspent
                                    if tx.output == self.public_key]

        self.coins.extend(coins_to_add)
        self.unspent_coins.extend(coins_to_add)

        # Remove the coins that were granted to this node in transactions that were removed from the blockchain.
        self.coins: List[TxID] = [coin for coin in self.coins if coin not in removed_txids]
        self.unspent_coins: List[TxID] = [coin for coin in self.unspent_coins if coin not in removed_txids]

    def remove_existing_chain(self, common_ancestor: BlockHash) -> List[Block]:
        """
        Remove the existing chain in the blockchain, starting from the next block after the given common_ancestor
        (i.e. starting from the block that its previous block hash is common_ancestor).

        :param common_ancestor: The block hash that will be the last block in the new blockchain.
        :return: A list of transactions that were removed from the blockchain (will be added to the MemPool later).
        """

        removed_chain: List[Block] = list()
        curr_hash: BlockHash = self.get_latest_hash()

        while curr_hash != common_ancestor:

            # Get the block that corresponds to the current BlockHash
            block: Block = self.blockchain[self.block_hash_to_index[curr_hash]]
            removed_chain.append(block)
```

```python

            self.blockchain.pop()

            self.block_hash_to_index.pop(curr_hash)
            for tx in block.get_transactions():
                self.txid_to_blockhash.pop(tx.get_txid())

            curr_hash: BlockHash = block.get_prev_block_hash()

        removed_chain.reverse()

        self.update_utxo(removed_chain)

        return removed_chain

    def append_new_chain(self, new_chain: List[Block], removed_transactions: List[Transaction]) -> List[Transaction]:
        """
        Append new chain to the end of the blockchain.
        This will also verify the validity of the blocks in the new chain,
        and truncate it if some block turned out to be invalid.
        This will also handle the UTxO set properly.
        :param new_chain: The new chain to add to the blockchain.
        :param removed_transactions: The previously removed transactions.
        :return: The updated list of removed transactions,
                 where a transaction is removed if it's contained in a new block.
        """
        for block in new_chain:

            transactions: List[Transaction] = block.get_transactions()

            # First of all, verify the "easy" stuff:
            # (*) The number of money-creation transactions is exactly 1.
            # (*) The total number of transactions is at most BLOCK_SIZE.
            amount_of_money_creation_is_valid: bool = (1 == sum(tx.input is None for tx in transactions))
            block_size_is_valid: bool = (len(transactions) <= BLOCK_SIZE)

            # Verify that all transactions are valid:
            # (*) The input transaction in the in UTxO.
            # (*) The signature is valid, i.e. was signed using the private key of the sender on the output + input.
            # (*) There is no contradicting transaction in the MemPool.
            # We exclude:
            # (*) Transactions that are money-creation (i.e. input is None).
            # (*) Transaction that already exist in our MemPool (it was verified when entering the MemPool,
            #     and the general validity check will fail because there is a contradicting transaction in the MemPool.
            transactions_are_valid: bool = all(self.verify_transaction_validity(tx, verify_with_mempool=False)
                                               for tx in transactions if tx.input is not None)

            # If this block is not valid, discard it and the rest of the chain.
            if not (amount_of_money_creation_is_valid and transactions_are_valid and block_size_is_valid):
                break

            # Remove all the transactions in the removed_transactions list that are in the current block,
            # because they don't need to enter the MemPool later.
            removed_transactions: List[Transaction] = [tx for tx in removed_transactions if tx not in transactions]
            self.add_to_blockchain(block)

        return removed_transactions

    def update_coins_according_to_new_chain(self, new_chain: List[Block]) -> None:
        """
        This function updates the balance allocated to this node according to a new block.

        :param new_chain: The new chain of blocks that was added to the blockchain.
        """
        transactions: List[Transaction] = [tx for block in new_chain for tx in block.get_transactions()]

        # coins_to_add are the coins in the new blocks in the blockchain that are assigned to this wallet.
        # coins_to_remove are the coins in the new blocks in the blockchain that this wallet used.
```

```python
        coins_to_add: List[TxID] = [tx.get_txid() for tx in transactions if tx.output == self.public_key]
        coins_to_remove: List[TxID] = [tx.input for tx in transactions]

        # Add the coins that were sent to this address, found in transactions in the blockchain
        # (in the relevant part of the blockchain, meaning from the last time we updated).
        self.coins.extend(coins_to_add)
        self.unspent_coins.extend(coins_to_add)

        # Remove the coins that were spent in transactions that made it into the blockchain.
        self.coins: List[TxID] = [coin for coin in self.coins if coin not in coins_to_remove]
        self.unspent_coins: List[TxID] = [coin for coin in self.unspent_coins if coin not in coins_to_remove]

    def get_length_of_tail(self, block_hash: BlockHash) -> int:
        """
        Get the length of the tail of the blockchain, starting from the next block of the given block_hash.
        :param block_hash: The block_hash that is not included in the tail
                           (the first block in the tail is the block with previous block hash equal to this block_hash).
        :return: The length of the tail (zero of the block already exists in the blockchain.
        """

        length_of_current_chain: int = 0
        curr_hash: BlockHash = self.get_latest_hash()

        while curr_hash != block_hash:
            length_of_current_chain += 1
            curr_hash: BlockHash = self.blockchain[self.block_hash_to_index[curr_hash]].get_prev_block_hash()

        return length_of_current_chain

    def get_common_ancestor(self, new_chain: List[Block]) -> BlockHash:
        """
        Get the common ancestor of the current blockchain and the given new chain.
        :param new_chain: The new chain to find the common ancestor.
        :return: the block-hash of the common ancestor.
        """

        common_ancestor: BlockHash = new_chain[0].get_prev_block_hash()
        return common_ancestor

    def notify_of_block(self, block_hash: BlockHash, sender: 'Node') -> None:
        """
        This method is used by a node's connection to inform it that it has learned of a
        new block (or created a new block). If the block is unknown to the current Node, the block is requested.
        We assume the sender of the message is specified, so that the node can choose to request this block if
        it wishes to do so.
        If it is part of a longer unknown chain, these blocks are requested as well, until reaching a known block.
        Upon receiving new blocks, they are processed and and checked for validity (check all signatures, hashes,
        block size, etc).
        If the block is on the longest chain, the mempool and UTxO set change accordingly.
        If the block is indeed the tip of the longest chain,
        a notification of this block is sent to the neighboring nodes of this node.
        No need to notify of previous blocks -- the nodes will fetch them if needed.

        A reorg may be triggered by this block's introduction. In this case the UTxO set is rolled back to the split point,
        and then rolled forward along the new branch.
        The mempool is similarly emptied of transactions that cannot be executed now.
        """
        new_chain: List[Block] = self.build_alternative_chain(block_hash, sender)

        # If the new chain is empty it means that the given block-hash was already in our blockchain,
        # and therefore no need to update anything.
        if len(new_chain) == 0:
            return

        common_ancestor_hash: BlockHash = self.get_common_ancestor(new_chain)
        length_of_current_chain_tail: int = self.get_length_of_tail(common_ancestor_hash)

        # Remove the existing chain, in order to reorganize the UTxO.
```

```python
        # This is needed in order to verify that each of the transactions in the new blocks uses an un-spent input.
        removed_orig_chain: List[Block] = self.remove_existing_chain(common_ancestor_hash)
        removed_orig_transactions: List[Transaction] = [tx for block in removed_orig_chain
                                                        for tx in block.get_transactions()]
        removed_orig_transactions_not_in_new_chain: List[Transaction] = self.append_new_chain(new_chain,
                                                                                              removed_orig_transactions)

        length_of_alternative_chain_tail: int = self.get_length_of_tail(common_ancestor_hash)

        if length_of_current_chain_tail >= length_of_alternative_chain_tail:
            # The alternative chain is not longer than the original one, so revert the changes.
            self.remove_existing_chain(common_ancestor_hash)
            self.append_new_chain(removed_orig_chain, removed_orig_transactions)

        else:
            # The alternative chain is longer than the original one, notify connections
            # about the new tip of the blockchain, and update the coins and the MemPool.
            self.notify_latest_block_to_all_connections()

            # Update the coins assigned to this node, both according to the removed chain of blocks,
            # and according the the new chain of blocks (only the block that actually entered the blockchain,
            # since some might have been discarded due to invalidity).
            self.update_coins_according_to_removed_chain(removed_orig_chain)
            self.update_coins_according_to_new_chain([block for block in new_chain
                                                      if block.get_block_hash() in self.block_hash_to_index])

            # Remove transactions that cannot be executed now from the MemPool.
            # This is done by trying to add the transactions to the MemPool (no need to notify the connections).
            transactions: List[Transaction] = self.mempool + removed_orig_transactions_not_in_new_chain
            self.clear_mempool()
            for transaction in transactions:
                self.add_transaction_to_mempool(transaction, notify_connections=False)

    def get_money_creation_transaction(self) -> Transaction:
        """
        This function inserts a transaction into the mempool that creates a single coin out of thin air. Instead of a signatu
        this transaction includes a random string of 48 bytes (so that every two creation transactions are different).
        generate these random bytes using secrets.token_bytes(48).
        We assume only the bank calls this function (wallets will never call it).
        """
        return Transaction(output=self.public_key, tx_input=None, signature=secrets.token_bytes(48))

    def add_to_blockchain(self, block: Block) -> BlockHash:
        """
        Append a new block to the blockchain, and handle the UTxO set accordingly.

        :param block: The block to append.
        :return: The block-hash of the block that was added to the blockchain.
        """
        block_hash: BlockHash = block.get_block_hash()
        transactions: List[Transaction] = block.get_transactions()

        self.block_hash_to_index.update({block_hash: len(self.blockchain)})
        self.blockchain.append(block)

        # Add the transactions in this block to the mapping to the corresponding block-hash.
        self.txid_to_blockhash.update({tx.get_txid(): block_hash for tx in transactions})

        # Add the new transactions to the UTxO.
        self.utxo.extend(transactions)

        # Remove the transactions in the UTxO that were spent in any of the transaction
        # in the transactions that were added to the blockchain.
        self.utxo: List[Transaction] = [tx for tx in self.utxo
                                        if tx.get_txid() not in [tx.input for tx in transactions]]

        return block_hash

```

```python
    def mine_block(self) -> BlockHash:
        """
        This function allows the node to create a single block. It is called externally by the tests.
        The block should contain BLOCK_SIZE transactions (unless there aren't enough in the mempool). Of these,
        BLOCK_SIZE-1 transactions come from the mempool and one additional transaction will be included that creates
        money and adds it to the address of this miner.
        Money creation transactions have None as their input, and instead of a signature, contain 48 random bytes.
        If a new block is created, all connections of this node are notified by calling their notify_of_block() method.
        The method returns the new block hash.
        """

        # The transactions that will be added to the blockchain are the first limit transactions in the MemPool.
        transactions_to_add: List[Transaction] = self.mempool[:BLOCK_SIZE-1]

        # Remove the transactions from the MemPool.
        self.mempool: List[Transaction] = self.mempool[BLOCK_SIZE-1:]

        # Add the money-creation transaction to the the block.
        transactions_to_add.append(self.get_money_creation_transaction())

        # Generate a new block and append it to the blockchain.
        block: Block = Block(previous_block_hash=self.get_latest_hash(), transactions=transactions_to_add)
        block_hash: BlockHash = self.add_to_blockchain(block)

        self.notify_latest_block_to_all_connections()

        self.update_coins_according_to_new_chain([block])

        return block_hash

    def get_block(self, block_hash: BlockHash) -> Block:
        """
        :param block_hash: The hash of the block to retrieve.
        :return: A block object given its hash.
                 If the block doesnt exist, a ValueError is raised.
        """
        index_in_blockchain: int = self.block_hash_to_index.get(block_hash)

        if index_in_blockchain is not None:
            return self.blockchain[index_in_blockchain]

        raise ValueError("The given block_hash does not exist in the blockchain.")

    def get_latest_hash(self) -> BlockHash:
        """
        This function returns the hash of the block that is the current tip of the longest chain.
        If no blocks were created, return GENESIS_BLOCK_PREV.
        :return: The last block hash the was created.
        """
        # If there are no blocks in the blockchain, return the hash of the genesis block.
        if len(self.blockchain) == 0:
            return GENESIS_BLOCK_PREV

        return self.blockchain[-1].get_block_hash()

    def get_mempool(self) -> List[Transaction]:
        """
        :return: The list of transactions that are waiting to be included in blocks.
        """
        return self.mempool

    def get_utxo(self) -> List[Transaction]:
        """
        :return: The list of unspent transactions.
        """
        return self.utxo

    def create_transaction(self, target: PublicKey) -> Optional[Transaction]:
```

```python
        """
        This function returns a signed transaction that moves an unspent coin to the target.
        It chooses the coin based on the unspent coins that this node owns.
        If the node already tried to spend a specific coin, and such a transaction exists in its mempool,
        but it did not yet get into the blockchain then the node should'nt try to spend it again until clear_mempool() is
        called -- which will wipe the mempool and thus allow the node to attempt these re-spends.
        The method returns None if there are no outputs that have not been spent already.
        The transaction is added to the mempool (and as a result it is also published to connected nodes).
        """
        if len(self.unspent_coins) == 0:
            return None

        selected_coin: TxID = self.unspent_coins.pop()
        transaction: Transaction = Transaction(output=target,
                                               tx_input=selected_coin,
                                               signature=self.private_key.sign(target + selected_coin))

        self.add_transaction_to_mempool(transaction)
        return transaction

    def clear_mempool(self) -> None:
        """
        Clears this nodes mempool. All transactions waiting to be entered into the next block are cleared.
        """
        self.mempool.clear()
        self.unspent_coins: List[TxID] = self.coins[:]

    def get_balance(self) -> int:
        """
        This function returns the number of coins that this node owns according to its view of the blockchain.
        Coins that the node owned and sent away will still be considered as part of the balance until the spending
        transaction is in the blockchain.

        :return: The number of coins that this wallet has (according to its view of the blockchain).
        """
        return len(self.coins)

    def get_address(self) -> PublicKey:
        """
        :return: The public address of this node in DER format (follow the code snippet in the pdf of ex1).
        """
        return self.public_key

    def __hash__(self):
        """
        This function is implemented to enable storing the Node object in hashable containers (such as a set).
        :return: The hash of this Node object.
        """
        return int.from_bytes(self.public_key, byteorder=sys.byteorder)

    def __eq__(self, other: 'Node'):
        """
        This function is implemented to enable storing the Node object in hashable containers (such as a set).
        :param other: Another Node to check equality to.
        :return: True if and only if the other node has the same public_key.
        """
        return self.public_key == other.public_key


"""
Importing this file should NOT execute code. It should only create definitions for the objects above.
Write any tests you have in a different file.
You may add additional methods, classes and files but be sure no to change the signatures of methods
included in this template.
"""
```