Blockchain and Digital Currencies

Homework 2: Block Chain

This homework is from the original assignment from the Princeton course website *Bitcoin and Cryptocurrency Technologies* at http://bitcoinbook.cs.princeton.edu/.

In this homework you will implement a node that's part of a block-chain-based distributed consensus protocol. Specifically, your code will receive incoming transactions and blocks and maintain an updated block chain.

Files provided:

`Block.java`	Stores the block data structure.
`BlockHandler.java`	Uses `BlockChain.java` to process a newly received block, create a new block, or process a newly received transaction.
`ByteArrayWrapper.java`	A utility file which creates a wrapper for byte arrays such that it could be used as a key in hash functions. (See `TransactionPool.java`)
`Transaction.java`	This is similar to Transaction.java as provided in Homework 1 except for introducing functionality to create a coinbase transaction. Take a look at `Block.java` constructor to see how a coinbase transaction is created.
`TransactionPool.java`	Implements a pool of transactions, required when creating a new block.
`UTXO.java`	From Homework 1.
`UTXOPool.java`	From Homework 1.

Files to be used from Homework 1:

`TxHandler.java`:

```
public class TxHandler {

    /** Creates a public ledger whose current UTXOPool (collection of unspent
     * transaction outputs) is utxoPool. This should make a defensive copy of
     * utxoPool by using the UTXOPool(UTXOPool uPool) constructor.
     */
    public TxHandler(UTXOPool utxoPool);

    /** Returns true if
     * (1) all outputs claimed by tx are in the current UTXO pool,
     * (2) the signatures on each input of tx are valid,
     * (3) no UTXO is claimed multiple times by tx,
     * (4) all of tx's output values are non-negative, and
     * (5) the sum of tx's input values is greater than or equal to the sum of
           its output values; and false otherwise.
     */
    public boolean isValidTx(Transaction tx);

    /** Handles each epoch by receiving an unordered array of proposed
     * transactions, checking each transaction for correctness,
     * returning a mutually valid array of accepted transactions,
     * and updating the current UTXO pool as appropriate.
     */
    public Transaction[] handleTxs(Transaction[] possibleTxs);
}
```

Create a public function `getUTXOPool()` in the `TxHandler.java` file you have created for homework 1 and copy the file to your code for homework 2.

When grading your code, we will run it with our reference `TxHandler.java`.

File to be modified:

`BlockChain.java`

```java
// Block Chain should maintain only limited block nodes to satisfy the functions
// You should not have all the blocks added to the block chain in memory
// as it would cause a memory overflow.

public class BlockChain {
    public static final int CUT_OFF_AGE = 10;

    /**
     * create an empty block chain with just a genesis block. Assume {@code genesisBlock} is
     * a valid block
     */
    public BlockChain(Block genesisBlock) {
        // IMPLEMENT THIS
    }

    /** Get the maximum height block */
    public Block getMaxHeightBlock() {
        // IMPLEMENT THIS
    }

    /** Get the UTXOPool for mining a new block on top of max height block */
    public UTXOPool getMaxHeightUTXOPool() {
        // IMPLEMENT THIS
    }

    /** Get the transaction pool to mine a new block */
    public TransactionPool getTransactionPool() {
        // IMPLEMENT THIS
    }

    /**
     * Add {@code block} to the block chain if it is valid. For validity, all transactions
     * should be valid and block should be at {@code height > (maxHeight - CUT_OFF_AGE)}.
     * For example, you can try creating a new block over the genesis block (block height 2)
     * if the block chain height is {@code <= CUT_OFF_AGE + 1}. As soon as {@code height >
     * CUT_OFF_AGE + 1}, you cannot create a new block at height 2.
     * @return true if block is successfully added
     */
    public boolean addBlock(Block block) {
        // IMPLEMENT THIS
    }

    /** Add a transaction to the transaction pool */
    public void addTransaction(Transaction tx) {
        // IMPLEMENT THIS
    }
}
```

The BlockChain class is responsible for maintaining a block chain. Since the entire block chain could be huge in size, you should only keep around the most recent blocks. The exact number to store is up to your design, as long as you're able to implement all the API functions.

Since there can be (multiple) forks, blocks form a tree rather than a list. Your design should take this into account. You have to maintain a UTXO pool corresponding to every block on top of which a new block might be created.

Assumptions and hints:

- A new genesis block won't be mined. If you receive a block which claims to be a genesis block (parent is a null hash) in the `addBlock(Block b)` function, you can return `false`.
- If there are multiple blocks at the same height, return the oldest block in `getMaxHeightBlock()` function.
- Assume for simplicity that a coinbase transaction of a block is available to be spent in the next block mined on top of it. (This is contrary to the actual Bitcoin protocol when there is a "maturity" period of 100 confirmations before it can be spent).
- Maintain only one global Transaction Pool for the block chain and keep adding transactions to it on receiving transactions and remove transactions from it if a new block is received or created. It's okay if some transactions get dropped during a block chain reorganization, i.e., when a side branch becomes the new longest branch. Specifically, transactions present in the original main branch (and thus removed from the transaction pool) but absent in the side branch might get lost.
- The coinbase value is kept constant at 25 bitcoins whereas in reality it halves roughly every 4 years and is currently 12.5 BTC.
- When checking for validity of a newly received block, just checking if the transactions form a valid set is enough. The set need not be a maximum possible set of transactions. Also, you needn't do any proof-of-work checks.

Your homework will be graded based on not only the implementation of the class but also the test suite to verify the implementation. Learn how to use *JUnit* from https://junit.org/junit4/.

Please submit your code together with a README.MD to your GitHub repository. For markdown (.MD) syntax see https://guides.github.com/features/mastering-markdown/.

Your REAMDE.MD needs to include 2 sections:

1. Your name and summary of the homework solution;
2. Design of the test suite: list each test function together with its description and purpose.