H2 Project: Block Chain

1. Overview & Analysis

The detailed requirements of homework 2 are in the `docs/Homework_2.pdf`.

- Overall, we will implement a **node** that's part of a block-chain-based distributed consensus protocol. Our code will receive **incoming transactions and blocks** and maintain an updated block chain.
- Specifically, we need to **implement** the `BlockChain` class, which is responsible for maintaining a block chain. Since the entire block chain could be huge in size, we should only keep around the most recent blocks. We also need to **do the testing**.
- Attention: Since **there can be (multiple) forks**, blocks form **a tree rather than a list**. Our design should take this into account. We have to maintain a UTXO pool corresponding to **every block** on top of which a **new block might be created**.

Based on the handling of transactions already implemented in homework 1, we started to build the block chain in depth. The block chain needs to be maintained and changed according to the established rules.

To complete this assignment, we must first analyze **the given class code**:

- **`UTXO.java`** and **`UTXOPool.java`** are same as the homework 1, which are used to manage the unspent transactions outputs. **`Crypto.java`** is also same as the one provided in homework 1.
- **`TxHandler.java`** has the same logic as my previous implementation. However, it simplifies the implementation of the `handleTxs()` function and does not take into account the unordered nature of the proposed transactions. It also **adds a new function `getUTXOPool()`** to it.
- **`Transaction.java`** is similar to `Transaction.java` as provided in homework 1 except for introducing functionality to create a `coinbase` transaction.
- **`ByteArrayWrapper.java`** is a utility file which creates a wrapper for byte arrays such that it could be used as a key in hash functions (this class is with `hashCode()` and `equals()` function implemented).
- **`Block.java`** stores the block data structure. It contains at least one transaction (the one called `coinbase` that received `COINBASE`), and the rest of the transactions are in a separate array `txs[]`.
- **`BlockHandler.java`** uses `BlockChain.java` to process a newly received block, create a new block, or process a newly received transaction.

There are also some more detailed information: The coinbase value is kept constant at 25 bitcoins whereas in reality it halves roughly every 4 years and is currently 12.5 BTC. So, the `Block.COINBASE = 25`.

Based on these elements, we need to finish **implementing and testing** the `BlockChain.java`. There are more detailed requirements:

- 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**.
- 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.

2. Implementation & Test

The runnable result code is a and b. In this section, I will elaborate the **implementation** and **test** code writing logic:

- The **implementation of BlockChain class**.
- The **test suite** to verify the related implementation.

2.1 Details of the implementation

Part 0. Storage structure of block chain.
Given there might be multiple forks, the data structure of the block chain **should be a tree rather than a list**. While, for the storage structure, there is no need to store the blocks in the tree because we can create a tree data structure using a list by storing the hash of this block and the state of this block in the block chain. The state of the block records the **1) content of this block**, **2) the height of the block** and **3) the corresponding UTXOPool**. Based on this, I build a `BlockState` class in `BlockState.java` to store these three elements and the ways to get them.

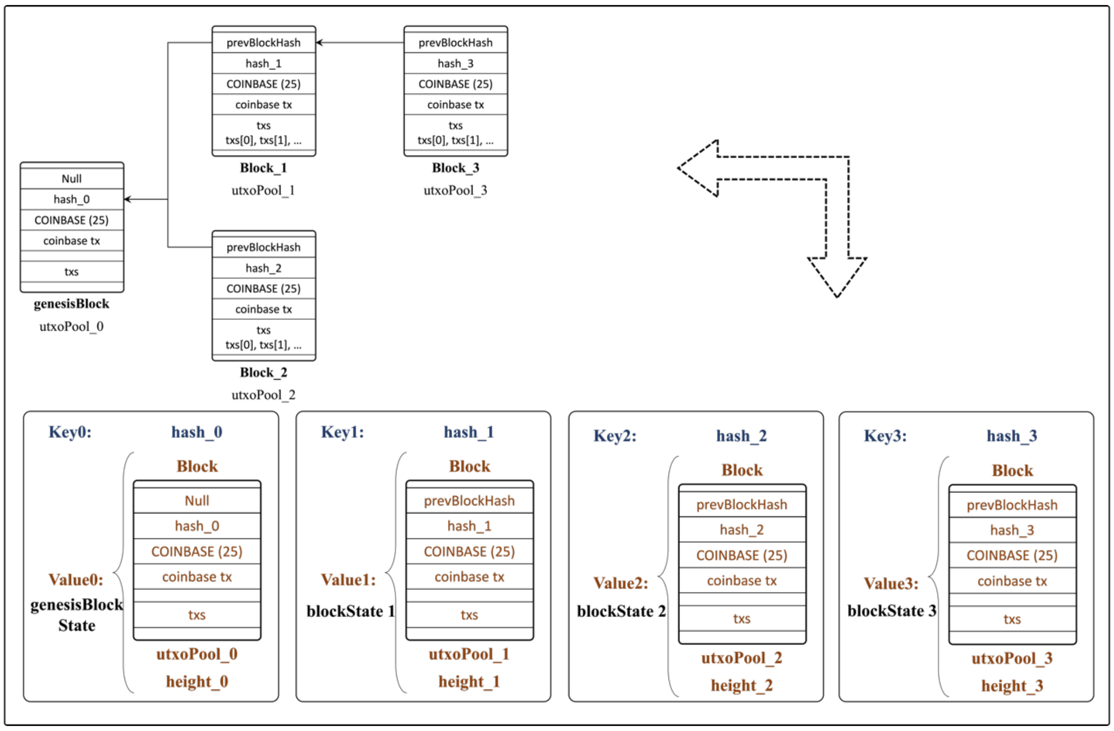


Figure 1. Store the tree-like data of block chain int map.

Considering that **the order of storage is meaningless**, it would be inconvenient if we use the list to store the block chain. Finally, I decided to **use Map as the storage structure**, as shown in Figure 1. The attribute in BlockChain should be defined as:

```
private Map<byte[], BlockState> blockChain;
```

The hash of the block is the key and the state of the blcok is the value in this map, and they can form a one-to-one mapping. At the same time, I maintain a state variable latestBlockState that indicates the status of the latest block in the longest valid branch. This not only allows for an **efficient implementation** of the data storage structure but also **improves the efficiency** of block chain related operations, such as searching.

Part 1. Construtor: BlockChain(Block genesisBlock)

This constructor is used to create a block chain with just a genesis block. The implementation can be devided into **four steps**:

- Step 1. Initialize the state of genesisBlock genesisBlockState, including the block, its height, and the current utxoPool.

```
Transaction coinbaseTx = genesisBlock.getCoinbase();
UTXOPool genesisUtxoPool = new UTXOPool();
UTXO utxo = new UTXO(coinbaseTx.getHash(), 0);
genesisUtxoPool.addUTXO(utxo, coinbaseTx.getOutput(0));
BlockState genesisBlockChainState = new BlockState(genesisBlock, 1,
genesisUtxoPool);
```

- Step 2. Add the genesisBlockState to an empty block chain.

```
blockChain = new HashMap<byte[], BlockState>();
blockChain.put(genesisBlock.getHash(), genesisBlockChainState);
```

- Step 3: Initialize the `latestBlockState` to record the latest block's state in the longest valid branch.

```
latestBlockState = genesisBlockChainState;
```

- Step 4: initialize the global Transaction Pool `transactionPool`.

```
transactionPool = new TransactionPool();
```

Part 2. Get Max Height Block State: `getMaxHeightBlock()` & `getMaxHeightUTXOPool()`

These two methods are used to get the **maximum height block** and the **UTXOPool** for mining a new block on top of max height block respectively. Since I have recorded the latest block's state in the longest valid branch, as long as the `latestBlockState` can be managed wisely, I can get them easily using the following interface:

```
public Block getMaxHeightBlock() {
  return latestBlockState.block;
}

public UTXOPool getMaxHeightUTXOPool() {
  return latestBlockState.utxoPool;
}
```

Attention: If there are multiple blocks at the same height, the `getMaxHeightBlock()` function should return the **oldest block**, so when faced with the same height, `latestBlockState` stores the state of the oldest block at all times.

Part 3. Get the Pool of Txs: `getTransactionPool()`

This method is used to get the transaction pool to mine a new block. From the assumptions and hints, we know that we could maintain **only one global transaction pool for the block chain**. So the interface should be:

```
public TransactionPool getTransactionPool() {
  return new TransactionPool(transactionPool);
}
```

Part 4. Add block to block chain: `addBlock()`

This method is used to add the block to the block chain if it is valid. It returns `true` only if the block meets the following three conditions:

- Condition 1. The block's parent node is valid. If the block claims to be a genesis block (parent is a null hash) or its parent node isn't in the previous block chain, returns `false`.

```
if (block.getPrevBlockHash() == null || !blockChain.containsKey(block.getPrevBlockHash())) {
    return false;
}
```

- Condition 2. All the transactions in the block should be valid. This condition makes sure that there is **no invalid transactions in the block**. Using `block.getTransactions()` to get the block's transactions and checking by `TxHandler().handleTxs()`, if the number of transaction is not change, then all are valid!

```
else if (!allTransactionsValid(block)) {
    return false;
}

private boolean allTransactionsValid(Block block) {
    // get the block's transactions
    Transaction[] blockTxs = block.getTransactions().toArray(new Transaction[block.getTransactions().size()]);
    // checking them by handler to get valid transactions
    BlockState parentBlockState = blockChain.get(block.getPrevBlockHash());
    Transaction[] validTxs = new TxHandler(parentBlockState.getUtxoPool()).handleTxs(blockTxs);
    // test whether number of Txs is change or not
    return blockTxs.length == validTxs.length;
}
```

- Condition 3. The CUT_OFF_AGE condition should be satisfied. For simplicity, the block should also meet the criteria that the `height > (maxHeight - CUT_OFF_AGE)`. If not, it returns `false`. Get the height of the block's parent node, check if `parentBlockState Height + 1 > maxHeight - CUT_OFF_AGE`.

```
else if (!CUT_OFF_AGE_Condition(block)) {
    return false;
}

private boolean CUT_OFF_AGE_Condition(Block block) {
    BlockState parentBlockState = blockChain.get(block.getPrevBlockHash());
    return parentBlockState.getHeight() + 1 > latestBlockState.getHeight() - CUT_OFF_AGE;
}
```

If the given block can meet the above three conditions, it returns `true`, which means the block is valid and can be added to the block chain. Then we need to do the following 3 steps to finish the `addBlock()` process:

- Step 1. Add this block into the block chain by recording its block state and the hash into the map `blockChain`.

- Step 2. Update the TransactionPool. From the assumptions and hints we know that it’s okay if some transactions get dropped during a block chain reorganization. So I don't take the block chain reorganization's effect into consideration and remove the block's transactions out of the TransactionPool.
- Step 3. Delete the previous block to meet the storage condition. Since the entire block chain could be huge in size, I just keep around the most recent blocks. The threshold value is represented as `STORAGE=20`.

The code of this part is long, so i don't paste it here, you can check it in the raw code file.

Part 5. Add Transactions: `addTransaction()`

This method is used to add a transaction to the transaction pool.

```
public void addTransaction(Transaction tx) {
  transactionPool.addTransaction(tx);
}
```

2.2 Details of the test suite

Here, a **full range of black-box tests** need to be performed on six functions implemented above. Overall, given the state of the blockchain can be divided into two parts: the single branch scenario and the forking scenario, I test them respectively. To make things easier, some preparatory work has been done before completing the tests in turn.

Preparations for tests `@Before`

First, to simulate a more realistic situation, I built a more secure keyPair generator `secureKeyPair()` that uses `SecureRandom()` to generate 2048-bit key. Second, I write a tool function called `createTransaction()` to help create transactions in quick way. Finally, I created the genesisBlock and initialized the BlockChain in the `@Before` module.

Tests for single branch scenario

Using the constructor to create a new empty block chain and then add some blocks to see if it can add the block to the blockchain correctly. By this process, the six functions can be test. The details are shown in the following table.

Test Function	**Test Purpose**	**Data for testing**	**The functions being test**

test_only_genesis_block()	Test for only genesis block existing.	No other data.	BlockChain()
			getMaxHeightBlock()
			getMaxHeightUTXOPool()
			getTransactionPool()

test_block_with_one_coinbase_tx()	Test for a valid block linked the genesis block. The valid block contains a transaction spending parent block's coinbase money.	A valid block with only one valid transaction.	All the six methods I have implemented.
test_block_with_several_valid_txs()	Test for a valid block linked the genesis block.	A valid block with three valid transactions.	All the six methods I have implemented.
test_block_with_invalid_txs_in_txPool()	Test for whether it can tell the invalid transactions in the transaction pool and leave them in the transaction pool rather than add them in the block.	A transaction pool with 2 valid transactions and an invalid transaction.	All the six methods I have implemented.
test_block_with_null_preHash()	Test if addBlock() can reject a block with null prehash.	A block whose previous hash is null.	All the six methods I have implemented.
test_block_with_wrong_preHash()	Test if addBlock() can reject a block whose prehash isn't in the blockchain.	A block whose prehash isn't in the blockchain.	All the six methods I have implemented.
test_block_with_invalid_txs()	Test if addBlock() can reject a block which contains invalid transactions.	A block whose prehash isn't in the blockchain.	All the six methods I have implemented.
test_storage_condition()	Test if the block will throw some blocks when the blockchain's max height doesn't satisfy the storage condition.	Genesis block and 23 valid blocks (The storage condition is 20 in my program).	All the six methods I have implemented.

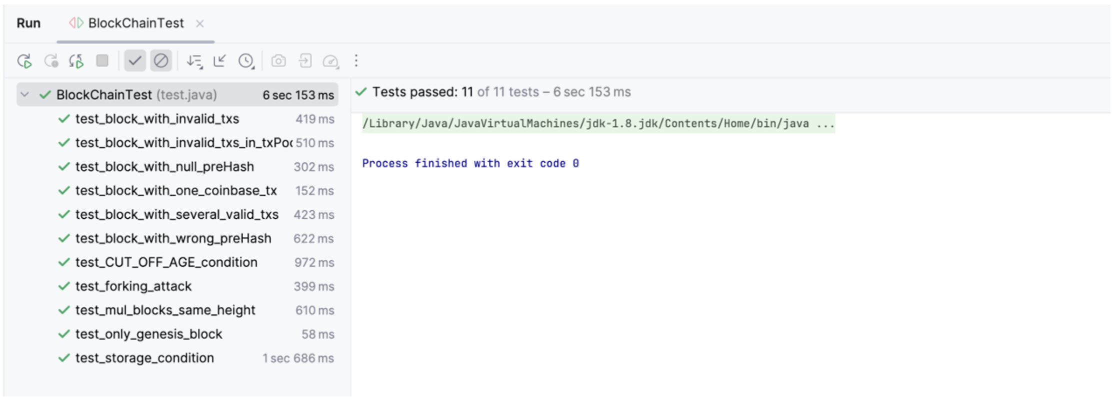
Note that when testing whether it can tell the invalid transactions in the transaction pool, I just test one case (double spending) because the other cases have been test in the last homework. So I just use the double spending as representative.

Test for forking scenario

In this scenario, I use the constructor to create a new empty block chain and then try to make some forks to see if it can add the blocks in the blockchain correctly.
All the cases can test all the six method I have been implemented so I won't mention this in the following table.

Test Function	**Test Purpose**	**Data for testing**
`test_CUT_OFF_AGE_condition()`	Test for CUT_OFF_AGE_Condition.	Genesis block, 11 valid blocks linked one by one, a block whose parent node is genesis block and a block whose parent node is the first block linked to the genesis block.
`test_mul_blocks_same_height()`	If there are multiple blocks at the same height, it should consider the oldest block in the longest valid branch. However, all of them should be in the clockchain.	Two valid blocks adding to the genesis block one after another.
`test_forking_attack()`	Test if there are two branches with the same height, whichever has the next block first will become to the new longest valid branch.	Genesis block and three valid blocks.

Test results in my environment
In my environment (details of which are described below), **all of the above 11 test functions passed**, as shown in Figure 2.



```
BlockChainTest (test.java)              6 sec 153 ms
    test_block_with_invalid_txs              419 ms
    test_block_with_invalid_txs_in_txPo(     510 ms
    test_block_with_null_preHash             302 ms
    test_block_with_one_coinbase_tx          152 ms
    test_block_with_several_valid_txs        423 ms
    test_block_with_wrong_preHash            622 ms
    test_CUT_OFF_AGE_condition               972 ms
    test_forking_attack                      399 ms
    test_mul_blocks_same_height              610 ms
    test_only_genesis_block                   58 ms
    test_storage_condition           1 sec 686 ms
```

```
Tests passed: 11 of 11 tests – 6 sec 153 ms

/Library/Java/JavaVirtualMachines/jdk-1.8.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

Figure 2. The test resuls.