PA 4: Hash Table and Triton Blockchain, 100 pts

Final Submission Due: Thursday, Nov. 8th, 2018 11:59 pm

Overview

1. In Programming Assignment 4, you will be implementing a Hash Table class.
2. You will then implement a Naive Blockchain called TritonBlockChain. It includes a Triton data class, Triton block class and a Triton blockchain class, using the knowledge of hashing.

For FINAL SUBMISSION, you will submit the following files:

- **HashTable.java** -- Provided for you as a starter file.
- **TritonData.java** -- Provided for you as a starter file.
- **TritonBlock.java** -- Provided for you as a starter file.
- **TritonBlockChain.java** -- Provided for you as a starter file.
- **README** -- Create by yourself

You will submit these files through autograder turnin script. Please refer to the submission instructions for earlier programming assignments on Piazza.

Do NOT CHANGE the following files in any way (and do not turn them in)

- **IHashTable.java** -- Provided for you as a starter file.

Grading for this assignment is broken down into the following parts:

- **HashTable.java** correct implementation -- **40 points** at final grading
- **TritonData.java** correct implementation -- **5 points** at final grading
- **TritonBlock.java** correct implementation -- **15 points** at final grading
- **TritonBlockChain.java** correct implementation -- **25 points** at final grading
- **README** correct explanation of the method you design -- **5 points** at final grading
- **Style** -- **10 points** at final grading

TOTAL: 100 Points

Starter Code Files

- **IHashTable.java**
- **HashTable.java**
- **HashTableTester.java**
- **Hash Functions.pdf**

- Sample hash functions.pdf
- TritonData.java
- TritonBlock.java
- TritonBlockChain.java
- TritonMiner.java
- transaction/ (folder)

Where to find the starter code

- From ieng6 server at
    - <accountname>@ieng6.ucsd.edu:/home/linux/ieng6/cs12f/public/pa4/
- Now you should have enough unix command line skills to copy the code from the public folder to your account, and securely transfer files between your ieng6 account and local environment.

Part 1 - Hash Table Implementation (40 points)

In this part, you will be implementing a basic Hash Table. We have provided you with an interface (IHashTable.java) and starter code that implements the given interface (HashTable.java). Fill in the blanks to complete the given methods that are listed below.

A. Implementing HashTable (60 points)

Method Name	**Method Description**	**Exceptions to throw**
insert(elem)	Inserts element elem in the hash table. Your program should return true or false, depending on whether it was inserted or not. Return true if item is inserted, false if it already exists.	NullPointerException if a null value is passed.
contains(elem)	Uses the hash table to determine if elem is in the hash table. Your program should return true or false, depending on whether the item exists.	NullPointerException if a null value is passed.
delete(elem)	Use the hash table to determine where elem is, delete it from the hash table. Your program should return true if the item is deleted, false if it can't be deleted (an item can't be deleted if it does not exist in the hash table).	NullPointerException if a null value is passed.
printTable()	Print out the hash table	none
getSize()	Returns the number of elements currently stored in the hashtable	none

Please note that to receive full credit for your Hashtable, your printTable() function should be **exactly** like in the format of:
<BUCKET NUMBER>: <Content1 in Separate Chain>, <Content2 in Separate Chain>

Sample output format for printTable():

```
0:
1: 10
2: 11, 1
3: 12, 2
4: 13, 3
5: 14, 4, 5
6: 15
7: 16, 6
8: 17
9: 18, 7
10: 19
11:
12: 9
13:
14: 8
```

Note that in the contains/delete method, you shouldn't need to search through the entire table to find an element.

Some implementation notes:

1. You may assume that all elements to be inserted are of type String.
2. For this assignment, use separate chaining to resolve collisions (You could use **Java's LinkedList**).
3. You must keep track of the load factor of your table and when the load factor becomes greater than ⅔, you must **double the size and rehash the values**. You must implement a private helper method rehash( ) to do this.
4. You can choose any hash function of your choice: you can use any of the functions we covered in class, in the book, or pick something from the provided list of hash functions. However, you must actually implement the algorithm in your HashTable class and **not** use java's in-built methods (i.e. String.hashCode()).
5. Make sure that the hash function you choose is fast, deterministic and uniform (mod the result by tableSize to avoid overflow)
6. Your rehash method should not take too long to insert and rehash. One recommendation is to use the given dictionary file as input to your hash table. Insert all the words in the dictionary into your hash table and make sure that it takes no longer than 30 seconds to load the entire dictionary.

<u>B. Keeping Statistics</u>

You should keep track of the following statistics. They should be reported by writing a line to a file every time the hash table is resized.

- ● The number of times you had to expand the table.
- ● The load factor in the table (report it before resizing), trimmed to 2 decimal places when written to the file. This value is reset whenever you expand the table.
- ● The number of insertions that encountered a collision (before resizing). This value is reset whenever you expand the table.
- ● The length of the longest known collision chain (before resizing). This value is reset whenever you expand the table.

1. Note that all except the first of these statistics will need to be reinitialized (reset or updated) whenever you expand the table.
2. Write it out to a text file in the following format r, c, n are integers, alpha is floating point number), each time you do a resize and rehash. Append (on a new line) a new set of values for each rehash (**r resizes, load factor alpha , c collisions, n longest chain)**
3. Notice the two constructors in HashTable class in the starter code provided. One takes in the initial size(any value > 0), while the other constructor takes in a filename as well as a size. The boolean 'printStats' is set to false by default, and is only changed to true in the constructor that receives a filename. Each time you expand and rehash your table, you must check if this boolean is set to true, and if yes, you must write the current statistics to the file before rehashing.
4. For example, after, say, 4 rehashes, the file will have the following lines:
   1 resizes, load factor 0.67, 1 collisions, 2 longest chain
   2 resizes, load factor 0.67, 7 collisions, 2 longest chain
   3 resizes, load factor 0.75, 55 collisions, 6 longest chain
   4 resizes, load factor 0.68, 221 collisions, 14 longest chain

5. There is no 'correct statistics' and your numbers will change depending on what hash function and initial table size you chose. These values will only indicate how good your chosen hash function is. Note that the values above indicate that our hash function is not that good and could be better, as we'd like to see the size of the longest chain remain relatively constant for the same load factor.

C. HashtableTester

For final submission, you will once again be graded on our master tester which we promise again that we will do our worst. Please note that the tester we provided in the starter code is extremely simplified. You should definitely add more unit tester to **HashTableTester.java** in order to push the limit of your Hashtable.

Part 2 - Triton Blockchain Implementation (60 points)

Now, for the first time in the CSE 12 history, you are going to implement a simplified blockchain. Your Blockchain should use hashing to ensure the security of the chain (we will talk about how to do that below).

A. Starting From Concepts

Nowadays, Blockchain, cryptocurrency, Bitcoin, … are such hot topics that even people who are not in the technology fields like talking about these fancy ideas. So what are the meanings of these words? In this section (and in discussion), we are going to briefly go over these concepts.

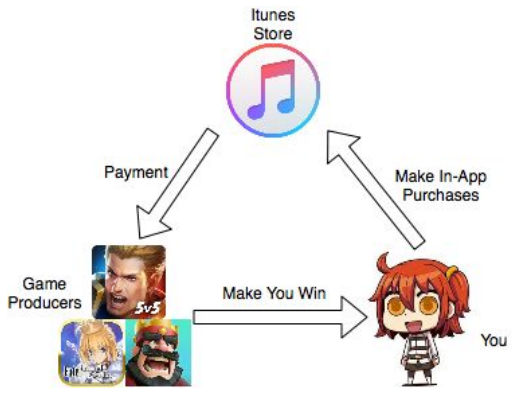
Blockchain

Blockchain, by design, serves as a decentralized, distributed data structure. You may see some definition online saying that "Blockchain is an open, distributed ledger", which is not wrong, but not accurate either. Blockchain involved in the use of cryptocurrency is a ledger, yet Blockchain itself may not contains "ledger" as content. This paragraph introduces you the definition of blockchain.

To get to know the significance and detailed structure of Blockchain, we need to understand the following concepts step by step:

Decentralization

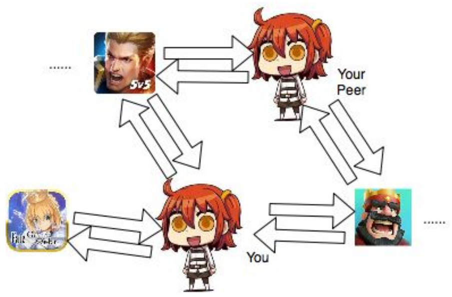
Firstly, let's consider a centralized process. Ex. You like pay-to-win in the mobile games and you decide to spend your lunch money this month in making some in-app purchases in your iPhone games (DON'T DO THAT! You need to eat lunch!). The trade procedure is: You make in-app purchases to iTunes Store -> iTunes Store notify the Game Producers, and pay them at certain time point -> Game Producers grant you the item or privilege you buy in order to win.



Graph 1. Centralized Trade Process

In this process, although you are buying the service from the game producer, the transaction actually involves a third party, the Itunes store. All of your transactions with all the game producers will be though the Itunes Store. Thus, If the Itunes service is down, your transaction will fail. Also, although you are just buying services from gamer producers, both you and the producers are providing information to a third party.

Thus, the ideal situation is a decentralized system, where you only interact with the seller, and if both of you claim the transaction is done, then the transaction is done.



Graph 2. Decentralized Trade Process

Imagine there are thousands of trade happening every pico-second, decentralization will save tons of resources, make the trade system autonomous and easier.

It's easier said than done. Without information centralization, how to ensure that both buyer and seller are not fraud? How to ensure the information in every trade is accurate? After decentralization, we need to ensure the credibility and the accuracy of the information. This brings up how blockchain uses its structure and hashing to ensure the security of the distributed data.

Triton Data Structure and Triton Block Structure

The term "blockchain" is pretty self-explanatory: A chain of blocks. The blocks, obviously, contains the information.

In the cases of cryptocurrency, the information stored is the ledger. In this programing assignment, you will implement the TritonData you stored into the blocks. TritonData mimics the ledger information in the following way:

TritonData Structure
List of Transactions: A List of transactions noted in the ledger

ProofId:	A proof of work you did to mine the block

The transactions will be in the format of strings. ProofId is an integer by which you prove you paid computation power to mine this block. (How to prove for work is explained in the “Proof of Work” Section below)

With the TritonData information we have in the format above, we put the information into the TritonBlock structure as below:

TritonBlock Structure	
Index:	The index of the block inside the blockchain
Timestamp:	The system timestamp when the block is created
Data:	The TritonData we want to save in the block
Prev_hash:	The hashing of the previous block in blockchain
Self_hash:	The hashing of the current block in blockchain

As you can see in the above structure, besides the TritonData, the TritonBlock also stores the hash value of the previous block, and itself’s hash value in the structure. Therefore, if the TritonData is affected by unstable network, and lost/flip a bit or two during transfer (or a hacker maliciously change the bits in another scenario), the stored data becomes inaccurate, then the prev_hash of the next block and the self_hash of the current block will not match up, invalidating the chain. That’s how the structure of the block makes the blockchain safer.

Triton Blockchain Structure

As stated earlier, each block is part of the blockchain. Therefore, the structure of the TritonBlockChain is easy as below:

TritonBlockChain Structure	
List of Blocks:	A list of TritonBlocks

The structure is intuitive and self-explanatory. Please noted that a blockchain is only meaningful if there is a block with information inside. Thus, unlike other list-like data structures, the constructor of the blockchain need to initialize a genesis block, which people will start mining from. This raises a new question: why do we mine more blocks on the blockchain.

Why We Mine More Blocks

As the creator of blockchain, the person will put ledger information in the genesis block. In order to let more people recognize the accuracy and the credibility of his ledger, the creator will want more block to be added into the list, where his ledger will be spreaded. But as miners, why do we bother spread the ledger info for him or her?

In order to attract miners, the creator usually says that "the next people who have my ledger information will have a piece of the fortune noted in the ledger!". Then all the people will be try to be the next block in chain because they can have reward by being miners. This is also why we have a static final variable **MINE_REWARD** in our TritonBlockChain class: the amount of the reward offered to the miner who owns the next block.

However, the reward raises another problem: now that all the people want reward, how can we decide who is next?

Proof of Work: How to Mine More Blocks

Now that everyone want a share in our ledger, who should we pass the ledger information to (since the blockchain is a list rather than a graph)? The solution is to **make the task of getting the ledger information really hard** so that we can distinguish people by some standard. The task described above will be **proof of work**.

The proof of work is always a complicated task, which will consume a lot of computational power of the miner's machine. And the task always grows harder and harder as the blockchain grows. How does machine spend computational power then? The simplest answer is just using loops. An example of proof of work is: Find the least common multiple of [the former proofId + 1] and a prime number (ex. 17) using loop.

The miners need to finish the complicated task before they can ask to create the next block in the blockchain. Since the computational powers vary from miner to miner, we prevent the scenario of all the miner ask to create the next block in chain.

In this assignment, you are allow to choose a computationally difficult task by yourself, as well as the hash function you used to hash the blocks (we will talk about easy and common ways to do these as well in discussion). Please write a **README file** contains the following information:

- **How are you hashing your blocks**
- **What is your proof-of-work task**

However, this strategy does not 100% eliminate the possibility of 2 miners completing the task at exactly the same time. What should we do? We will only discuss this situation in discussion (for fun), since your naive TritonBlockChain only runs locally, and you are also the sole miner on your local machine.

Cryptocurrency

After the description above, you already know that the information stored in the blocks in chain can be anything. You do not have to put the information of legal money or existing item transactions in the blocks. You can just put “I have 100 Saint Quartzs!” in the data section of the block. If the value of “saint quartz” is publicly recognized, miners start to use the computation power of their machine to mine your “saint quartz”, and people are buying “saint quartz” using legal money, “saint quartz” actually become a valuable cryptocurrency.

Bitcoin

Bitcoin is one of the many types of popular cryptocurrency. Bitcoin’s actual structure, hashing, and proof of work is much more complicated than the TritonBlockChain we implemented this time. If you are interested in digging more, here are some keywords for your own researches: Bitcoin Core, SHA-256, and nonce.

B. Implementation Details

Your TritonData, TritonBlock, and TritonBlockChain classes should implement methods listed as below

TritonData

Method Name	Method Description	Exceptions to throw
TritonData()	Constructor of triton data	none

List<String> getTransactions()	Get all transactions	none
int getProofId()	Get proof id	none
String toString()	Return a string in the following format Eg: “ DATA Start------------------------------- Proof of work: XX Transaction 0 Transaction Content: name1 name2 X$ Transaction 1 Transaction Content: nam3 name4 y$ Transaction 2 ... Transaction Content: Triton coin earned: 1 DATA End ------------------------------- ”	none

(Notice: You can get the amount of triton coin earned through MINE_REWARD variable inside TritonBlockChain class)

TritonBlock

Method Name	Method Description	Exceptions to throw
TritonBlock(int index, long timestamp,	Constructor that initializes all the data	none

TritonData data, String prev_hash)		
String hashBlock()	Hash the data (choose the algorithm you like)	none
int getIndex()	Return index	none
String getTimestamp()	Return timestamp	none
TritonData getData()	Return block’s data	none
String getPrev_hash()	Return previous block’s hash	none
String getSelf_hash()	Return cur block’s hash	none
String toString()	Return a string in the following format Eg: “ TritonBlock k Index: k Timestamp: 1540589541260 Prev Hash: 1016750814 Hash: 1150849736 (TritonData info) ”	none

TritonBlockChain

Method Name	Method Description	Exceptions to throw
TritonBlockChain(int index, long timestamp, TritonData data, String prev_hash)	Constructor. 1. Initialize a local blockchain 2. On initialization, add a genesis block using the information in parameters.	none
TritonBlock makeNewBlock(TritonBlock lastBlock, TritonData newData)	Makes the next block. 1. Initialize the index by the former block's index + 1 2. Initialize the time stamp: a. Hint: You can do this by calling *System.currentTimeMillis();* 3. Initialize the prev_hash by getting the self_hash of previous block 4. Use the TritonData and above variables to initialize a new TritonBlock and return it.	none
boolean beginMine(List<String> curTransactions)	If curTransactions is empty, return false Otherwise 1. Get a proof id generated by proofOfWork 2. Append a string recording the number of triton coin earned to curTransactions in the following format: a. "Triton coined earned: <REWARD_NUM>" 3. Create a TritonData object with proof_id and curTransactions 4. Create a new TritonBlock with TritonData 5. Add TritonBlock to current blockchain	none

int proofOfWork()	find number that is divisible by both lastProofId and 13 (This function is used to prove cpu usage. In real blockchain, it will be much more complicated)	none
Boolean validateChain()	Check the validity of the current blockchain by checking if prevBlock’s hash matches current Block's prevHash.	none
List<TritonBlock> getBlockChain()	Getter of blockchain	none
String toString()	Return data is the following format: “ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ T R I T O N B L O C K C H A I N ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ TritonBlock 0 (Block data...) ”	none

Full Sample output

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
T R I T O N   B L O C K C H A I N
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TritonBlock 0
Index: 0
Timestamp: 1540592055917
Prev Hash: 0
Hash: -1037253988
DATA Start--------------------------------
```

```
Proof of work: 1
DATA End  --------------------------------

TritonBlock 1
Index: 1
Timestamp: 1540592055918
Prev Hash: -1037253988
Hash: -915589582
DATA Start--------------------------------
Proof of work: 26
Transaction 0
Transaction Content: mary bob 10$
Transaction 1
Transaction Content: mary bob 12$
Transaction 2
Transaction Content: Triton coined earned: 1
DATA End  --------------------------------

TritonBlock 2
Index: 2
Timestamp: 1540592055919
Prev Hash: -915589582
Hash: 307556879
DATA Start--------------------------------
Proof of work: 351
Transaction 0
Transaction Content: simon tian 3$
Transaction 1
Transaction Content: tian ming 3$
Transaction 2
Transaction Content: Triton coined earned: 1
DATA End  --------------------------------
```

(Note: block 0 is genesis block, thus it has prevHash 0, index 0, and no transaction recorded. Starting at block 1, there will be transaction in each block)

C. To run your blockchain

We've provided you TritonMiner.java, which you can compile, run, and produce the above output. It is also a playground for you to modify and try out your own TritonBlockChain implementation. The correct implementation would result in same output format as above.

D. Testing

For final submission, since we already give you really detailed function signatures and explanations in the above sections. We will do strict unit testing for functions you are not designing by yourself (which is Hashing and ProofOfWork), and manually examine your README explanations on how you design hashing and ProofOfWork. We do not provide you simple JUnit tester class, but you are encouraged to write your own.

Turning in your code

Look through your programs and make sure you've included your name, ID and login at the top of each of them.

Files

Along with your name and cse12fxx account name at the top of each program you wrote, you also need the appropriate comments at the top of your HashtableTester.java file. We are expecting you to hand in the following files:

- **HashTable.java**
- **TritonData.java**
- **TritonBlock.java**
- **TritonBlockChain.java**
- **README**

How your assignment will be evaluated

- **Coding Style (10 points)**
    - Does your class and tester properly generate javadoc documentation
    - Is your code readable by others
        - consistent indentation (Please use SPACES, not tabs)
        - are variables sensibly named
        - are access modifiers (public, private, protected) used appropriately
        - are helper methods used when needed to reduce code duplication

- **Correctness**
    - Does your code compile?
    - Does it pass all of your unit tests that you defined?
    - Does it pass all of our unit tests? (we are users of your ADT implementation, you don't get access to our unit tests)
        - We'll check basic functionality, including Exceptions
    - Does your code have any errors? (e.g. generates exceptions when it isn't supposed to) (That would be bad).

- **Unit test coverage (Optional but Highly Encouraged)**
    - Have you created at least 5 more meaningful unit tests? (that's a bare minimum, you are very likely to have significantly more tests) Does it detect errors in a reasonably buggy implementation of HashTable or TritonBlockChain?
    - Does your unit testing approach for HashTable appear to be sufficient? We supply you with one unit test for HashTable(), that is not sufficient. Good tests will include testing what happens before the head and after the tail, as well as more specific tests for each of the methods.

- We’re not telling you exactly what to test, that’s part of learning to create approaches to developing and debugging more complicated code. We will test your tester against reasonably buggy implementations of HashTable. In addition, your tests should help you develop your code and insure its proper operation.

Keep up the great work, and hack like a champion!

Reference:

We want to give credit to every hacker who provides insights on designing this part of the assignment. They are:

Le Wang - LaiW3n
Rohit Thotakura
Mohit Mamoria