Project 3

DUE: October 29th at 11:59pm

Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Have a style (indentation, good variable names, etc.) and pass the provided style checker (see P0).
- Comment your code well in JavaDoc style and pass the provided JavaDoc checker (see P0).
- Have code that compiles in your user directory without errors or warnings.
- For methods that come with a big-O requirement, make sure your implementation meets the requirement.
- Have code that runs (see detailed commands below).

You may:

- Add additional methods and class/instance variables, however they **must be private**.
- Create inner/nested classes, however they **must be private**
- Import these classes only: `File`, `FileNotFoundException`, `MessageDigest`, `StandardCharsets`, `Iterator`, `Scanner`

You may NOT:

- Use arrays of any type anywhere in your program.
- Make your program part of a package
- Add additional public/protected/package-private methods or variables
- Use any Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, etc.)
- Alter any class/method signatures defined in this document or the template code
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements)
- Add any additional libraries/packages
- Create any additional classes or interfaces
- Use lambda expressions

Setup

- Download the **`p3.zip`** and unzip it. It contains a template for all the files you must implement

Submission Instructions

1. Make a new temporary folder and copy there your **`.java`** files. Do not copy the test files, jar files, class files, etc.
2. Upload the temporary folder to OneDrive as a backup (this is not your submission, just a backup!)
3. Follow the Gradescope link provided in Blackboard>Projects and upload the files from your temporary folder onto Gradescope's submission site. Do not zip the files or the folder.

Testing

This project requires extensive testing on your part. Allow at least a couple of days just for testing. You don't need to write unit tests but you must cover many corner cases which can become very tedious when you work with linked structures and trees/recursion in particular.

A small example of how to run your program is provided with the file **`P3.java`**. This is not a tester; just an example of how to execute your program. Do not submit this file. You can use it anyway you want.

A "compliance checker" will be added to Gradescope next week. This is not a tester; it just checks for class/method signatures having the correct names and parameters. Passing the compliance checker doesn't mean that your code is correct neither that the compilation with the actual autograder won't fail.

Grading

Grading will be divided into two portions: unit testing (80%) and manual inspection (20%). Code that doesn't compile will get zero points (no manual inspection in this case). Only your last submission will be graded.

Unit testing

`MerkleTree`	25 pts
`SinglyLinkedList`	15 pts
`PriorityLine`	10 pts
`Block`	10 pts
`Blockchain`	10 pts
`Transaction`	2 pts (most of the code is provided)
`Utilities.loadTransactions`	3 pts
`Utilities.veriryTransaction`	5 pts
`Order`	0 pts (provided)

Manual grading

Big-O requirements	8 pts
Code reuse	7 pts
Javadoc documentation	3 pts
Code style compliance	2 pts

Penalties

Hard coding	Zero points for the method that uses hard coding
Non-permitted imports	Zero points for the method that uses the imported class
Other kinds of violations	Zero points for the method where the violation occurred

Extra credit for early submissions

- 1% extra credit rewarded for every 24 hours your submission made before the due time. Max reward is 5%
- Your latest submission will be used for grading and extra credit checking. You can't choose which one counts

Overview

In this project we're going to implement a simplified version of the blockchain data structure that is used by many cryptocurrencies including Bitcoin. Disclaimer: We only consider the data structure part here, not the networking, the consensus mechanism, etc. And still, many details have been removed, abstracted or conveniently modified to accommodate the learning goals of this course. This is **not** the actual blockchain. A blockchain is, simply put, a linked list of blocks where each block contains a linked list of financial transactions (see Figure 1).

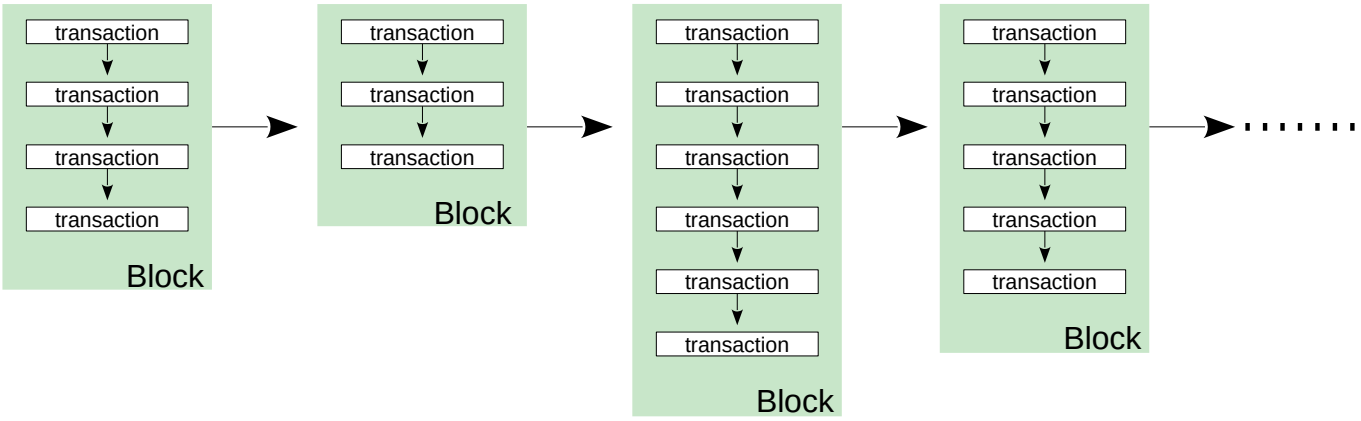

Figure 1: Blockchain is a list of blocks where each block contains a list of transactions

The transactions are generated in real time 24x7 from all over the world and are sent to a queue in order to be processed/cleared and added to a block in the blockchain. Each transaction must pay a small fee for the processing but there are no restrictions on the amount of this fee (i.e. the person that initiates the transaction can pay whatever they want). On the other hand, the company that will process the transactions and generate a block has an incentive to first process the transactions with the higher fees and leave the ones with lower or zero fees for later. Therefore, the queue that collects the transactions is not a FIFO but a priority queue, where the priority is based on the amount of the fee paid. The higher the fee one pays the earlier their transaction will be cleared and added to the blockchain. What happens in practice is that the processor (called *miner*) starts dequeueing transactions from the priority queue and, when the cumulative amount of fees reaches a certain threshold, it packages them into the next block to be added to the blockchain (see Figure 2).

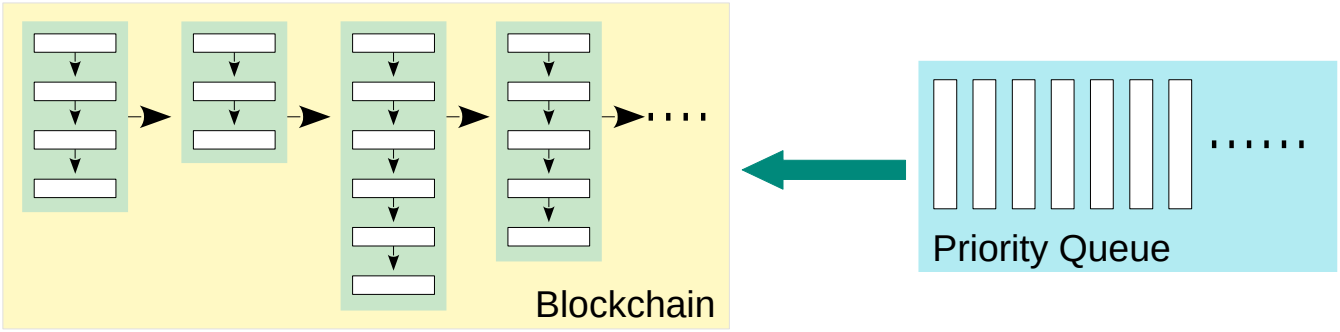

Figure 2: A priority queue feeds the blockchain with transactions

This means that, a) the blocks will usually vary in size, i.e. will contain a different number of transactions, b) a transaction paying a very low fee can wait many blocks before it's processed, and c) the transactions within a block are not ordered by the time of their creation but rather by the amount of the fee they paid.

Adding transactions to the blockchain is not our only task though. We also want to be able to lookup and verify any past transaction before we allow someone to spend the funds they got from it. This is because, unlike our banking accounts, the blockchain doesn't keep track of our balance; it only keeps a record of the transactions themselves. So, when someone tries to spend a previously received amount, the only means to tell if they truly possess this money is by searching the past transactions in the blockchain. One way to do this lookup and verification process is to go through every single transaction in the history of the blockchain, but as you can imagine this is very inefficient and slow. What we're doing instead is to build, for each block in the blockchain, a very compact and "smart" summary of its transactions and store it in the respective block so that, later, when someone tries to verify a transaction, we don't go through the entire list of transactions but we only check this "smart" summary. In a nutshell, this "smart" summary is the root of a binary tree called Merkle tree (see Figure 3). Because the root in this case is generated in a bottom-up fashion (i.e. the leaves are the inputs and the root is the output), it can encode in a compact form enough information for every transaction in the block.

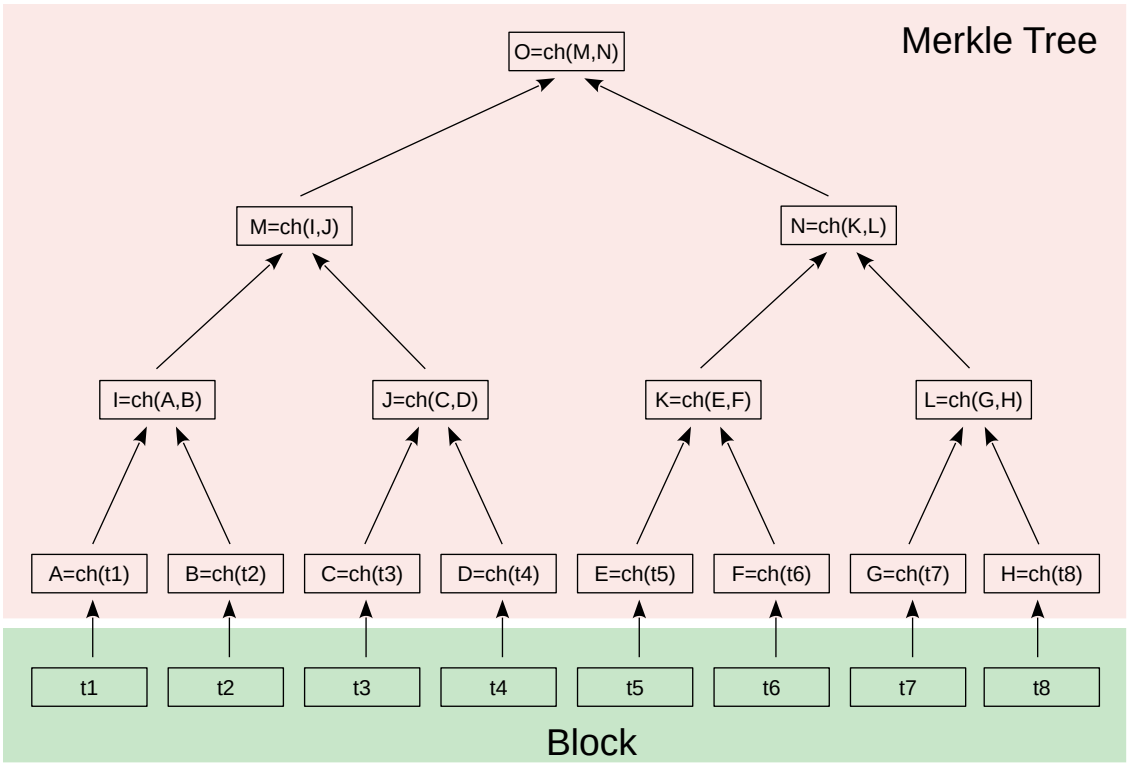

Figure 3: Construction of a Merkle tree for a single block in the blockchain

To build this reversed binary tree we use a special function called **cryptographic hash** function denoted as ***ch(·)*** in the figure. This is not the same as the hashing we discussed in class. Unlike the hash functions we used in class, a cryptographic hash function has the nice property that produces a **fixed-size hash code** that is **practically unique**. So, if we take a transaction, e.g. **t1**, convert it to a String with the `toString()` method and then pass it through the ***ch(·)*** function, the result will be a unique hash code. Designing a good cryptographic hash function is a difficult task and definitely not the goal of this course. Therefore, we're going to use a built-in method, namely the SHA-256, which generates a 32-bytes long hash code. In the provided Utilities class we have included a method that takes a String of any

length (it can even be a whole textbook) and returns the SHA-256 hash code as a String with a fixed size of 64 characters. These characters represent HEX values (be reminded that 1 byte is 2 HEX values) and the reason we use HEX instead of Unicode is to avoid the situation where the value of a byte is a non-visible character. Here are some examples of running the provided method:

input parameter (`String`)	**return value (`String`)**
Mason	`9e00106e1ab0502ff078f66c66b35fa9cc86a41753b5e7b1aea90bc29284eb23`
MasoN	`a44d0dcc8e331c2a4127ffc302f90f875fbff6a8abce36dce628de25f825469b`
George Mason University	`45174c91a28566053f1b2ecb40a01a80d7239544c7320b233a8ad1423e205a95`

Notice that the outputs for `Mason` and `MasoN` are completely different despite the similarity of the two inputs. Three things are important here and you must always keep them in mind:

1) No matter what the size of the input is, the resulted hash code will be fixed at 32 bytes (64 HEX characters)

2) The slightest change in the input will generate a completely different output, i.e. it's impossible to guess the return value if you know the return value of a similar input

3) You can assume that the hash code is practically unique, i.e. it's extremely unlikely that two different inputs, no matter how close they are to each other, will generate the same hash code.

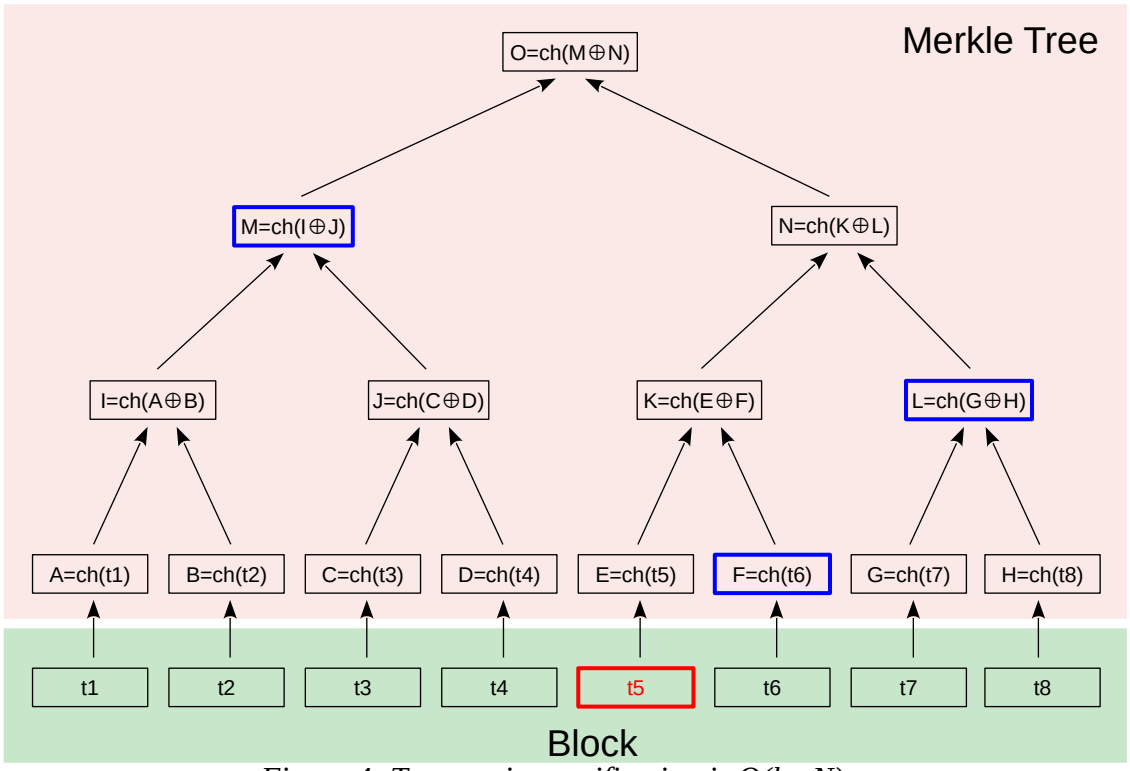

Figure 4: Transaction verification is O(logN)

If we build the Merkle tree of a block, then we don't need anymore the go through the entire list of its transactions when we want to verify a certain transaction. See Figure 4 for an example. Assume that we want to prove that a certain transaction, let's call it $\overline{\texttt{t5}}$, is contained in the block at the location of **t5**. To prove that, we need to provide only the hash codes **F**, **L**, **M** that correspond to the pairs we will need as we build the root of the tree. Then the verifier will generate the root of the Merkle tree by applying the cryptographic hash function incrementally and will check if the resulted root is the same with the one stored in the block. More specifically, it will first generate $\overline{\texttt{E}}=\texttt{ch}(\overline{\texttt{t5}})$, then it will use this along with the provided **F** to generate $\overline{\texttt{K}}=\texttt{ch}(\overline{\texttt{E}} \oplus \texttt{F})$, then it will use this along with the provided **L** to generate $\overline{\texttt{N}}=\texttt{ch}(\overline{\texttt{K}} \oplus \texttt{L})$, then it will use this along with the provided **M** to generate $\overline{\texttt{O}}=\texttt{ch}(\overline{\texttt{N}} \oplus \texttt{M})$, and finally it will compare the produced $\overline{\texttt{O}}$ with the **O** that is already stored in the block. If these two are equal, it verifies that the provided $\overline{\texttt{t5}}$ is indeed the transaction `t5` contained in this block at this exact location. The ⊕ operator in all these calculations is the XOR function, but you don't need to implement it yourself; in the Utilities class we provide two versions of the cryptographic hash function, one with a single parameter and one with two parameters.

This whole process of verification has a time-complexity which is equal to the height of the Merkle tree. Since the tree is **binary** and **perfect**, the complexity is `O(logN)` which is way faster than searching all the nodes of the list in `O(N)`. The reason the Merkle tree is a **perfect tree** is because we need to apply a paired hashing at every level of the tree. In case the leaves of the tree (i.e. the number of the transactions in the block) are not an exact power of two, we use right padding – we assume that there are "dummy transactions" at the right end of the list. We don't actually add them to the list, we just imagine them being there and we use them to generate the root of the Merkle tree. When you want to apply the cryptographic hash function on a "dummy transaction", you will simply hash the string **DUMMY** since there is not an actual object to apply the `toString()` method on.

Classes

Implement the following classes in the order provided. Make sure you test each class thoroughly before you move to the next one. The `MerkleTree` is the most challenging one, but none of the other classes depends on it (i.e. you can get full credit for the other classes even if `MerkleTree` is not perfect).

`Utilities`

It contains three auxiliary methods that help with the rest of the project. One is the cryptographic hash function which is fully implemented in the template (it has two versions depending on the number of parameters you want to use). The second one is a method for loading transactions from a text file that you must implement yourself. And the last one is a method for verifying transactions which you must implement yourself as well.

`Transaction`

This class represents an individual financial transaction that the blockchain can hold. Most of the code is provided already, you just need to infer and complete the missing parts. Make sure you do not modify the `toString()` method because the returned value will be used as input to the `cryptographicHashFunction`. Transaction objects must be comparable and the comparison is based on the fee that the transaction paid. The higher the fee the "larger" the object. You will need this comparison when you enqueue transactions into the priority queue.

`SinglyLinkedList`

Implements a singly-linked list data structure. It has the basic operations of a linked list and it's also iterable. The node must be implemented as a private nested class. The iterator must be implemented as an anonymous inner class.

`PriorityLine`

Implements the priority queue data structure. You must use the `SinglyLinkedList` as the underlying structure. It has the basic operations of a queue and it's also iterable.

`Block`

This class represents an individual block in the blockchain. It uses a `SinglyLinkedList` to store all the transactions contained in the block. It also holds a copy of the root of the respective Merkle tree in order to allow efficient verification of individual transactions without going over the entire list of transactions contained in the block. The class is iterable.

`Blockchain`

This class represents the entire Blokchain data structure. It holds a `SinglyListList` of `Block` objects. All the work is mainly in the constructor that reads in the transactions from the priority queue and builds the whole chain. The class is iterable.

`MerkleTree`

Implements a data structure that represents a Merkle tree as depicted in Figure 3. The tree must be stored as a linked structured – you may not use arrays of any type. The node must be implemented as a private nested class.

`Order`

An enumeration that has three constants representing the three types of depth-first traversal. The file is provided; no need to edit it.