Project 2:
Block Chain
Authenticated Storage

DUE: Tuesday, March 7 at 11:59pm Extra Credit Available for Early Submissions!

BasicProcedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided)
- Have a style (indentation, good variable names, etc.) and pass the automatic style checker (see P0).
- Comment your code well in JavaDoc style AND pass the automatic JavaDoc checker (see P0).
- Have code that compiles with the command: **`javac *.java`** in your user directory without errors or warnings.
- For methods that come with a big-O requirement (check the provided template Java files for details), make sure your implementation meets the requirement.

You may:

- Add additional methods and class/instance variables, however they **must be private**.
- You may use only the ArrayList data structure from the java.util package

You may NOT:

- Make your program part of a package.
- Add additional public methods or public class/instance variables, remember that local variables are not the same as class/instance variables!
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Alter provided classes that are complete (**`Hashing and SHA512`**).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the `p3.zip` and unzip it. This will create a folder **`section-yourGMUUserName-p2`**;
- Rename the folder following the same instructions from P0, P1, and P2.
- Complete the `readme.txt` file (an example file is included: **`exampleReadmeFile.txt`**)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip **`section-username-p2.zip`** (no other type of archive) following the same rules for **`section`** and **`username`** as described above.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Project Description

In this project you will implement an authenticated data structure for storing files on a server. The data structure you will be using is a doubly linked list.
Authenticated linked lists are the basic building block of block chain and cryptocurrency.
For instance, assume you have a collection of files on your laptop and you want to store it in the cloud (remote servers that provide storage services to clients). Your system should allow the realization of the following operations:
• **addFile**: adds a new file to the cloud server
• **getFile**: retrieve some file stored in the cloud server
• **deleteFile:** deletes the last file on the cloud server

you can easily implement this system using a (doubly) linked list on the cloud servers where each file is stored in a linked list node. For simplicity, a file could be represented as a String object in our project. Storing new files consists of inserting a new node at the end of the linked list. To retrieve a particular file, the system searches for it in the linked list (if present) and returns it to the client.
So far, no authentication mechanism to ensure that the files stored on the cloud are not maliciously corrupted or tampered is implemented. If any unauthorized modification is applied on the linked list in the cloud, the client should be able to detect it. Accordingly, we need to implement one more operation that we term as: **verifyIntegrity** to allow the client to make sure that the files they stored are not modified or tampered with.

Introduction to Crypto Hash Functions (CHF)
Cryptographic hash functions (CHF) are special hash functions used for developing security protocols. What distinguishes Cryptographic hash functions from traditional hash functions is that they are resistant to collisions. Technically when using cryptographic hash functions, it is computationally-infeasible to find two messages producing the same hash value. This renders the crypto hash value a unique identifier ("fingerprint") of a particular input message. Any small modification on the message (even a single-bit change) should result in a different output from the crypto hash function. This property makes these functions very practical in developing secure authentication and integrity algorithms in cryptography.
We will utilize the CHF and the doubly linked list data structure in order to create fingerprints for the client stored files in the cloud.
For each file stored in the cloud, there corresponds a node in the linked list containing that file (a String object). In addition to that, we will also store a fingerprint (digest) at each node of the linked list. Accordingly, each node of the linked list now includes the file as well as a small digest.

How is the Digest Computed?

Every time you add a node to the linked list, you will store two items: the String of data (representing the file) along with a small message digest.
The digest is generated by applying the CHF on two items: the file data $\boldsymbol{file_i}$ concatenated with ***'&'*** concatenated with the digest of the previous node $\boldsymbol{digest_{i-1}}$. Therefore:

$$\boldsymbol{digest_i} = \text{CHF}(\boldsymbol{file_i} + \text{'}\boldsymbol{\&}\text{'} + \boldsymbol{digest_{i-1}})$$

As you might notice, the first node containing the first file String has no previous node ($digest_{i-1}$), lets assume $digest_0$ is certain string ***startDigest***. For this reason, this first node's digest $digest_1$ will be generated by apply CHF on the file String $file_1$ concatenated with '&' concatenated with ***startDigest:*** $digest_1 = CHF(startDigest + '\&' + file_1)$

Therefore, the first node in the linked list with contains: $file_1$ and $digest_1$
Every node in the linked list contains two items: $file_i$ and $digest_i$
Figure 1 demonstrates the wuthenticated doubly linked list for our file storage.

1. Now to add a new file $file_2$ to the linked list, ***addFile*** will first check whether there was any tampering or change done on the linked list (**verifyIntegrity** discussed in point 3). If no tampering is detected, then a new node is created where the data part is set to $file_2$ and the digest $digest_2$ is set to $CHF(digest_1 + '\&' + file_2)$. We will store $digest_2$ (the last digest computed) as an instance variable at the client ***proofCheck*** to be used in **verifyIntegrity**.

2. To delete a file from the cloud server, we will delete the last node in the linked list. Before deleting the node, the integrity check is similarly done to ensure that there was no tampering done on our authenticated linked list (call **verifyIntegrity).** We will update ***proofCheck*** to be the last digest present in the linked list. Figures 3 and 4 demonstrate deleting and inserting into the linked list.

3. To ensure integrity, the client calls **verifyIntegrity** which checks the authentication of the linked list by using the ***proofCheck*** and the linked list from server. The client accesses the linked list nodes one by one starting from the last node and computes the following:
   Lets say we have ***n*** nodes:
   - client checks whether ***proofCheck*** is equal to $digest_n$ if not it will return false
   - If true, for i=n to 2:
     - Checks that $digest_i = CHF(digest_{i-1} + '\&' + file_i)$ if not it will return false.
   - Finally it checks if $digest_1 = CHF(startDigest + '\&' + file_1)$

   - If all checks pass it will return true otherwisse it will output false.

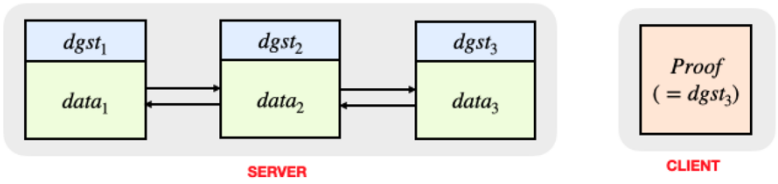

Figure 1: Authenticated doubly linked list demonstration.

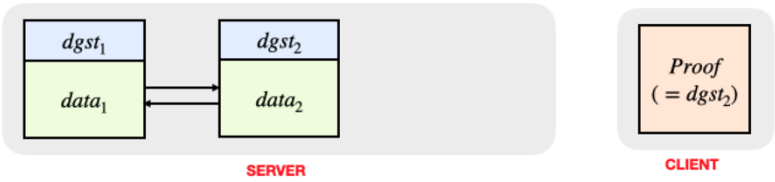

Figure 2: The same doubly linked list after deleting file 3.

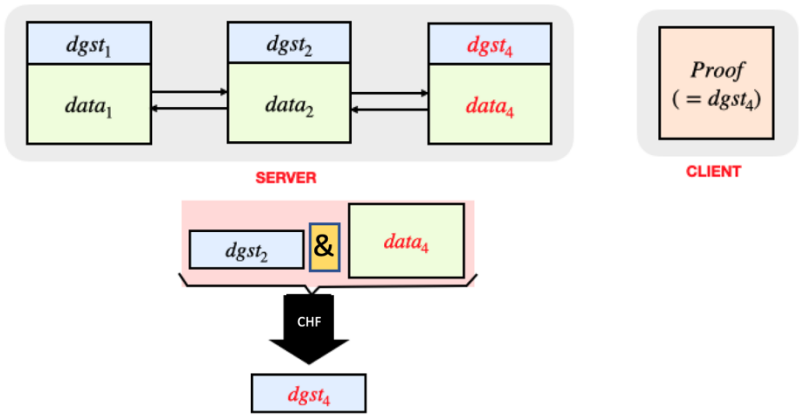

Figure 3 : Adding $file_4$ with $data_4$ after deleting $file_3$ in the authenticated doubly linked list.

Authenticated Lists

• *public class Node:* Represents the nodes to be stored in the linked list
- *public Node* ***previous****:* reference to a Node object.
- *public Node* ***next****:* reference to a Node object.
- *public String* ***digest****:* a string that will contain the digest.
- *public String* ***file:*** the string representing the file.

• *public class AuthDLList* :Represents the authenticated doubly linked list containing the following attributes and methods:
- *public Node* ***head****:* represents a reference to the first node present in the authenticated list.

- *public static final String **startDigest**:* a string containing the starting digest for all authenticated linked lists
- *public Node **Tail**:* Node reference to the last node in the linked list
- *public static boolean **verifyIntegrity**(AuthDLList currentList, String proofCheck) throws IntegrityCheckFailedException*: returns True if all the nodes in currentList are checked and validated and if *current.lastNode.digest = proofCheck* otherwise raises *IntegrityCheckFailedException.*
  For example for each Node u the following should hold in order to render the list valid and no exception thrown: (CHF corresponds to Hashing.cryptHash)

u.digest= Hashing.cryptHash(AuthDLList.startDigest + "&" + u.file) if u.previous = null
or Hashing.cryptHash HF (u.previous.digest +"&"+ u.file) if u.previous != null

where Hashing.cryptHash is a static method to generate the Hexadecimal representation of the cryptographic hash of a String s.

- *public String **insertFileNode**(String data, String check) throws IntegrityCheckFailedException:* this method takes as input a *String data*, and a *String check*. Checks the integrity of the list using the *String check*, creates a new node and inserts it at the end of the list with file corresponding to data, and updates the digests of all the nodes such that the updated list is valid. Returns the digest of the last node of the updated list.
  - *First check that the linked list is not changed. This is done using verifyIntegrity (this is where the string check is utilized).*
  - *If the above check passes, then you create a new object of class Node by setting it's different instance variables: previous, next (set these appropriately), file set to data and set the digest string using the Hashing.cryptHash on the appropriate input.*
  - *Finally, the above Node object is added at the end of the list, and the proofCheck is updated appropriately(the digest of the last node).*

- *public String **deleteLastFile**(String check) throws EmptyDLListException, IntegrityCheckFailedException:* this method takes one input the check string. It checks the integrity of the list utilizing the String check, deletes the last node of the list, and updates all the nodes digests accordingly. Returns the digest of the last node after the delete is applied on the list.
  - *First checks that there is no tampering, and this is done using verifyIntegrity (using the check string).*
  - *If the previous check passes, then the method successfully deletes the last node of the AuthDLList. Should update the Tail attribute.*
  - *Finally, it returns the new proofCheck string.*
- *public String **deleteFirstFile**(String check) throws EmptyDLListException, IntegrityCheckFailedException:* This method is similar to DeleteLastFile method. First, it checks the integrity of the list by utilizing the check string. It deletes the first node and updates the digests of all the nodes such that the updated list is valid. Returns the digest of the last node as the new proofCheck of the updated list.

- *public static Node **retrieveNodeFile**(AuthDLList current, String check, String file) throws IntegrityCheckFailedException, FileNotFoundException*: checks

the integrity of the list by utilizing String check, returns the first Node u of all the nodes v present in the AuthDLList current having u.file =file. If there is no such node u then raises FileNotFoundException

- *It first checks the list using verifyIntegrity.*
- *If the check passes, it starts with the first node in the doubly linked list (that is, head), and checks if the node's file value is same as the file string attribute. If so, it returns this node, else it moves to the next node.*
    - *If no such node is found, it throws FileNotFoundException.*

• *public class Hashing extends SHA512*: this class extends from the SHA512 class and is responsible of generating the CHF values needed to generate the digest values of the linked list nodes. The important method in this class is:

- *public static String cryptHash(String s):* this is a static method to generate the Hexadecimal representation of the cryptographic hash of a String s.