CS 25100 (12283,LE2), Data Structures and Algorithms, Fall 2016

Project 3: File Verification with Merkle Trees

- **Part 1:** Task 1, Due: Saturday, October 22, 11:59pm
- **Part 2:** Tasks 2 and 3, Report, Due: Saturday, October 29, 11:59pm
- **Bonus Part:** Task 4, Due: Saturday, October 29, 11:59pm
- **Late Penalty**: 20% per day. A maximum of 1 late day (including slip days) for Part 1. A maximum of 2 late days (including slip days) for Part 2. No late submission allowed for the Bonus part.
- **Skeleton Code**: project_3_skeleton.zip
- **Data**: Updated Data Files

Overview

Merkle Trees (also called hash trees) are commonly used to detect inconsistencies in data as they allow secure and efficient verification of data stored remotely. They are widely used in peer-to-peer systems (e.g., Ethereum and Bitcoin), in protocols (e.g., BitTorrent and Apache Wave), and in cloud and database systems (e.g., Apache Cassandra, Riak, and Amazon DynamoDB). They are named after Ralph Merkle, a computer scientist working in the area of cryptography who patented the work.

In this project, you will build Merkle Trees and use them for file verification. In a simple cloud based system, the **Client** uploads a file onto the **Server**. The Client wishes to periodically verify the contents of the file on the Server (typically, to check if it reflects the changes made). To verify this, the Client can download the entire file and check it. However, this can involve a significant amount of data transfer. Merkle tress allow one to verify a file with minimum data transfer.

The basic idea underlying Merkle trees is to represent a file using hashed values of its content organized as a tree. The Client determines this tree, maintains only the value associated with the root of the tree, sends the file to the server, and deletes the file after verifying the upload. Using the received file, the Server determines the same tree and stores both the file and the tree. To verify that the upload was successful, the Client does not request the entire file from the Server. Instead, the Client requests a certain portion of the tree. The amount of data sent will be small compared to the file size and it will allow the Client to verify that the file is stored correctly and is intact. If verification fails the client can compute which blocks are the cause of the failure and transfer those blocks again.

The next sections define Merkle trees, describe how they are built and how file verification works. Make sure you understand all these concepts and are able to explain how a Merkle tree works before starting your implementation.

Building a Merkle Tree

A Merkle tree is a complete binary tree (see slide 18 from week 4.2); i.e., it is a binary tree in which every level, except possibly the last level, is completely filled, and all nodes are as far left as possible. Note that Merkle trees have the same shape as heaps. Consequently, Merkle trees can be stored in arrays using the same index computations used in heap operations. **Your implementation must have the root at index 1**.

To build a Merkle Tree M, the file is divided into blocks of size 1024 Bytes. Say, this process creates m blocks, B_1, B_2, ..., B_m. The Merkle tree built will have m leaves and m-1 interior nodes. Leaf 1 will be the leftmost leaf on the level closest to the root. The numbering of the leaves is as shown in Figure 1: the tree has 11 leaves with 5 leaves on level 3 and 6 leaves on level 4.

The values at the nodes of tree M are determined as follows:

- **Values at the leaves.** Leaf i corresponds to block B_i, $1 \le i \le m$. The value at leaf i, Val[i], is the hashed value of block B_i; Val[i] = Hash(B_i). Leaf 1 in a Merkle tree is the leftmost leaf on the level closest to the root (see Figure 1).
- **Values at the internal nodes.** For internal node v, Val[v] is generated from the values at its two children a and b: Val[v] = Hash(Val[a]+ Val[b]), where + stands for string concatenation, a is the left child and b is the right child. Thus, the values associated with each child are concatenated and hashed to form the value of the parent.

The above described process leads to a bottom-up computation of the values at nodes which terminates once the value at the root has been determined. The value at the root of the tree is called the *Master-Hash*. A Merkle tree is stored in an array. The indexing used to determine the index position of the parent and the children are done as in a heap.

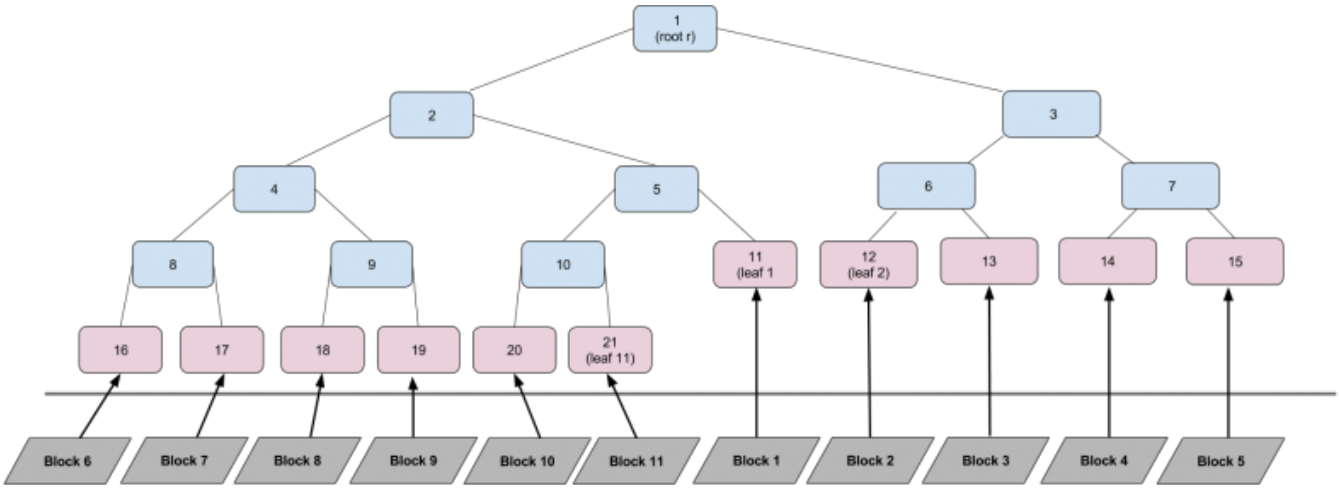

Figure 1: 21-node Merkle tree

Figure 1 shows a 21-node Merkle tree. The tree is shown above the horizontal line. The blocks are shown below the line and their mapping to the leaves nodes is indicated by the arrows. The internal nodes are formed by concatenating and hashing the children.

The setup preceding the file verification has the following steps:

- The Client partitions the file into blocks of size 1024 Bytes and builds the Merkle tree (as described above).

- The Client sends the file to the Server.
- The Client retains the Master Hash of the tree and records the number of leaves. The file will be deleted if verification passes.
- The Server, after receiving the file, computes the Merkle tree of the file (in the same way that the Client computed). The Server retains both the Merkle tree and the file.

Thus, at the end of the Setup phase:

- The Client has the Master-Hash of the file along with the number of blocks
- The Server has the file and the Merkle tree with the Master-Hash.

Verification

The Client initiates a verification to verify the contents of the files stored by the Server.

Path Siblings: For any node v in a tree, consider the path from v to root r. The *path siblings* on the path from v to root r are the children of the nodes on the path that are **not** on the path and the node v itself. For example, for the tree shown in Figure 1, the path siblings from node 21 to the root are nodes 21, 20, 11, 4 and 3 as illustrated in Figure 2. Note the **order** of the path siblings: it starts with the node v.

The verification process proceeds as follows:

1. The Client selects the index of any node in the tree except the root node. Let p be this index.
2. The Client sends integer p to the Server.
3. The Server, upon receiving indes p, sends **the nodes that are the path siblings on the path from p to the root** to the Client. Using the path siblings, the client is able to compute the Master Hash.

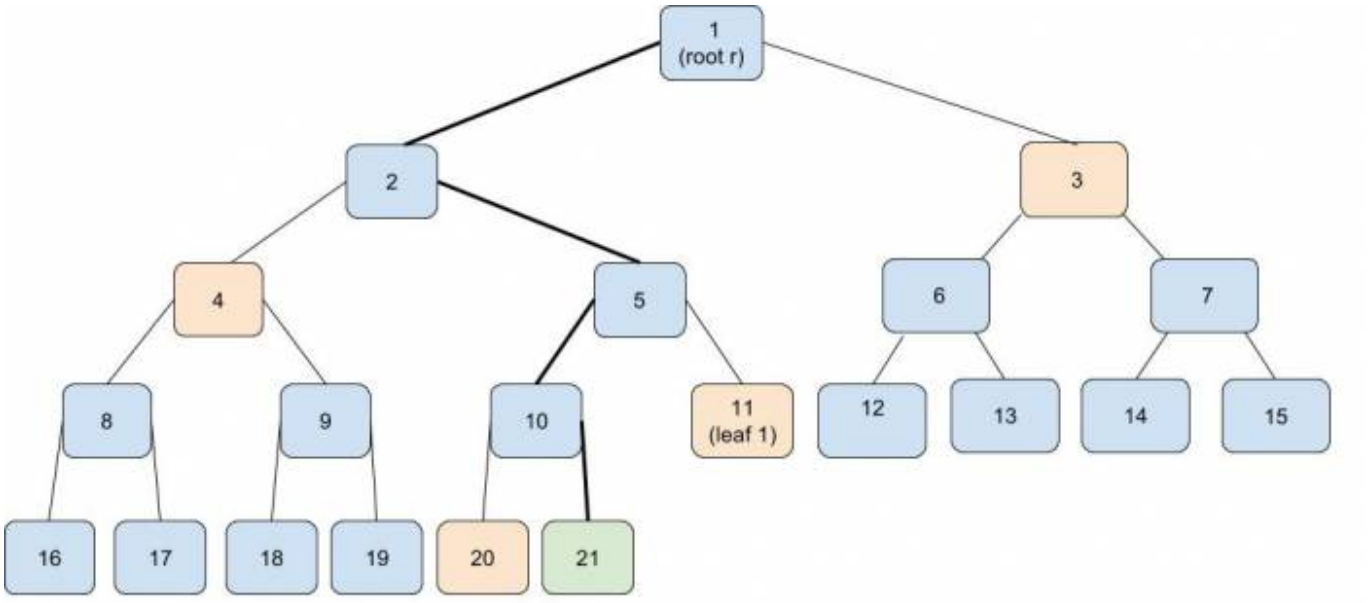

Figure 2: Challenge-Response with node 21 (path siblings in yellow)

The verification process is considered a *Challenge-Response*. The Client "challenges" the Server by sending the index of a node and the Server "responds" by replying with a portion of the Merkle-tree (i.e, the path siblings and their values).

Consider again the tree shown in Figure 1. The Client starts a verification with leaf node 21.

- Index 21 is sent to the Server as “Challenge”.
- The Server “responds” by sending the Client the nodes 21, 20, 11, 4 and 3, along with their values; i.e., (21,val(21)), (20,val(20)), (11,val(11)), (4,val(4)), (3,val(3)). The data structure is specified in the skeleton code.
- Using the information about the nodes received, the Client computes the Master-Hash.
- If the computed Master-Hash matches with the Master-Hash that the Client has stored, the verification is a success.

Success indicates that the file at the Server is intact. The Server was able to prove to the Client that the file is intact. In addition, the amount of data transferred was small. Typically, the Client sends multiple challenges. To complete the description of how a Merkle tree is built, we next address the hash functions used when computing the values stored at the nodes.

Hash Functions

We use hashing to build the Merkle Tree. In this project, we use two hash functions:

1. A Cryptographic hash function, **MD5**. MD5 produces a 128-bit hash value and is a very widely used hash function. It is used as a checksum to check file corruptions.
2. A weaker hash function, **SumHash**. See Week 6.2, Slides 7 and 8, for a description.

Merkle trees derive their power and effectiveness from the strength of cryptographic hash functions. Using cryptographic hash functions, each file results in a unique tree and no two files can have the same tree. Proper verification of the file fails in case of collisions.

Implementation Details
Each of the two hash functions *MD5.java* and *SumHash.java* are of type *HashFunction* and contain the following methods:

- method *hashBlock* which is used at a leaf. It takes a String input and returns a String.
  - in MD5 it returns the MD5 hash
  - in SumHash it must return the sum of bytes cast as integers (ignoring overflows) as a String.
- method *concatenateHash* which is used at the internal nodes. It takes the left *Node* and right *Node* as input and returns a String hash.

You are only expected to implement the *hashBlock* method in *SumHash.java*, everything else has been implemented in the skeleton.

When building the Merkle Tree, read a file block by block. Apply *HashFunction.hashBlock* to each block of the file to generate the hashes at the leaves. For all interior nodes, apply *HashFunction.concatenateHash* to the child nodes (left followed by right).

Tasks

Task 1: Build a Merkle Tree

Build the Merkle tree using MD5 as the hash function and the code provided to split the file into blocks.

Classes to implement: Merkle Tree

This class is the implementation of a MerkleTree. You need to implement the function *build*, which reads the input file block by block and creates the tree. You will have access to the field *tree* which is of type *Node[]* and is the array representation of the tree. A basic template for reading the file block by block and padding the last block is provided in the skeleton code.

You are supposed to use *Configuration.hashFunction.hashBlock()* at the leaves and *Configuration.hashFunction.concatenateHash()* at the internal nodes to get the values of hash. The function should return the *masterHash* as a String on completion. A simple way to do this is to access the root node and return *Node.getHash()*

Note: Like heaps, the root node must be at index 1. For any node, the parent is i/2 and the children are 2*i and 2*i + 1. For internal nodes, concatenation is always left+right not vice versa. Therefore use *Configuration.hashFunction.concatenateHash()* to generate the concatenated hash of two child nodes.

To test your code you can run: *java merkle.Process -i <inputfile>*
This will return a Master Hash. You can also view the whole tree if you run the above command with a *-v* flag

Task 2: Challenge-Response

One Challenge-Response consists of two subtasks.
1. Given the index of a (non-root) node in the Merkle tree, determine the path siblings.
2. Given the path-siblings, compute the Master-Hash.

We will provide an implementation of *MerkleTree.java* two days after the Part 1 deadline. For Task 2, you can choose to use ours or your own implementation of *MerkleTree.java*.

Classes to implement:

Server This class models a server. You need to implement *generateResponse* which takes the block number and returns the path siblings as a *List<IMerkleTree.Node>*. You can use functions provided by *this.merkleTree.getNode* to access nodes.

Client This class models a client. You need to implement *verifyResponse* which takes the output of *Server.generateResponse()* and computes the master hash and returns a boolean true if it matches the Master Hash of the uploaded file. To generate a hash you need to use *Configuration.hashFunction.concatenateHash()* and finally you need match it with *this.masterHash*.

To test your code use *java merkle.Process -i <inputfile> -c 10*
This builds the Merkle tree, runs verification on 10 randomly selected nodes in the tree, and outputs the number of times verification passed. All verifications should pass. You can view the whole tree and detailed output if you run the above command with a -v flag.

Task 3: SumHash

Implement *SumHash* hash function. Then, use the hash function to build Merkle trees for all the files provided in the data set. One of the files given in the data set has a wrong Master Hash value in the SumHash column in the table. Report which file this is. If more than one file gives MasterHash different from the table, your code contains a bug.

You need to implement the *hashBlock* method of *SumHash*. You can use standard String functions like *getBytes*. Note that casting bytes to int is as simple as *int a = (int) aByte*.

To test your code you can run: *java merkle.Process -i <inputfile> -c 10 -hf sum*
Apart from building the merkle tree using the *SumHash* hash function, this will also run verification on 10 random nodes in the tree and output the number of times verification passed. You can also view the whole tree and detailed output if you run the above command with a -v flag

Task 4: Bonus Task

Implement class *CollisionGenerator*: Given is a Merkle Tree of 31 nodes generated using *SumHash*. Using the 31 hash values, generate a file which corresponds to the given Merkle tree. This process is called spoofing and illustrates why strong hash functions are important. Note that you do not know the original file.

Class *CollisionGenerator* takes a serialized Merkle Tree as input and creates a byte sequence (representing the file) by reverse engineering SumHash. This file is then dumped to disk. You need to implement *generateCollision* which takes *merkleTree* as input and writes to *file*. A basic template for writing to the file block by block is provided.

In summary, you are asked to (1) create a file from the given Merkle tree by reverse engineering , (2) generate a Merkle tree for the file you created, and (3) check if the Master hash matches the given one.

To test your code you can run:
java merkle.Process -t spoof -i <merkle tree file> -hf sum -o <file to generate>

This page contains ReverseEngineer.merkletree.txt which is supposed to be the input merkle tree and the masterhash which is the expected master hash.

Skeleton Code Structure

The skeleton code provided contains a folder (*implementation*). You must only edit files in this folder. Do **NOT** edit the files outside this folder. All the functions that have been provided to you are outside

this folder and you can use them as described in the tasks above.

You will see that files in *implementation* extend some other classes, for example: *implementation.MerkleTree* extends IMerkleTree. This means:

- You can use all variables and functions declared in the parent class.
- You cannot change the signature of a method declared in the parent class. Changing signatures will lead to compilation failures!!!

Classes in *implementation* may have some commented code or functions implemented. This code is to help you out. **Again, only the files inside the *implementation* folder are to be modified.**

Tests

Every task has instructions on how to test. The dataset also has the master hash values provided so that you can verify your code. Your Merkle Tree works if it returns the same master hash as provided. Success in these tests may not guarantee a full score. There will be no AutoGrader points for code which doesn't compile and pass the basic tests.

Note: We recommend using Configuration.print() to print objects. This way you can control printing debug output by using the -v flag.

Running the Code

The main function is in the file *Process*. You can run the whole process via commandline as follows:

- Go to the folder in which *merkle* folder is present and compile the files using javac merkle/*.java.
- Run the program using java merkle.Process -h and follow the usage instructions
- We've copied the instructions here for you:

```
usage: java merkle.Process [options]
-i is a required option
    -i <path to input file>
    -hf <hash function> [default md5]
        md5 for MD5
        sum for SumHash
    -b <block size> [default 1024]
    -c <verify count> [default 10]
    -v
        for verbose output
    -h or --help
        help
```

- To generate a merkle tree
  - *java merkle.Process -i /path/to/file*
  - This will print the master hash of the Merkle Tree
- To generate and verify a merkle tree
  - *java merkle.Process -i /path/to/file -c 10*
  - This will print the master hash of the Merkle Tree

    - Then it will try the challenge and response on 10 random nodes in the tree and print how many times it was successful
- To generate and verify a tree using SumHash
    - *java merkle.Process -i /path/to/file/Merkle_tree* ***-hf sum***
    - Same as above but now it'll use SumHash
- To print debug firendly output append ***-v*** to the command
- To change the block size to 2048 for example append ***-b 2048*** to the command
- To change the number of times verification runs change the number after *-c*

Note: By default the hash function is set to *md5*, the block size is *1024*. You always need to pass the input file as shown in the examples above. Students who solve the bonus task need to run the main file with ***-h*** and figure out the usage on their own.

Data Set

Data files and their Master Hashes are available here. Use the files to test your code.

Analysis Questions

Your **typed** report must answer each question briefly and clearly. Data presented in any experimental analysis should be presented in tables or charts. All material must be readable and clear and prepared in a professional manner. For charts and tables that can easily be produced with Excel (or equivalent software), handwritten material **will not** be accepted.

Experimental Analysis

Unzip the file (10mb.zip); file 10MB.data has size 10MB. Generate Merkle trees using the MD5 hash function when the file is partitioned into blocks of sizes 32, 64, 128, 256, 512, 1024, and 2048 bytes and record the time taken for generating each tree. Repeat the experiment with this (25mb.zip) file. Plot the block size on x-axis and the time taken on the y-axis (in milli seconds) for both the files. Discuss and explain any patterns you observe.

Analytical Questions

Assuming that applying a hash function takes O(1) time. Answer each of the following questions in a clear and precise manner. Express time bounds in terms of m.

1. Given a file that is divided into m blocks, what is the asymptotic worst-case time of building the Merkle tree from the m blocks? Explain your answer.
2. Given a leaf node u in the Merkle tree, what is the asymptotic worst-case time of determining the path siblings of leaf u? Explain your answer.
3. For a Merkle tree M containing m leaves, what is the asymptotic worst-case time of performing one Challenge-Response given a node in tree M? Explain your answer.
4. Let r be the root of a Merkle tree M (with m leaves) and let u its left child. Assume the server “lost” the hash values of nodes on the path from the left child of u to the leftmost leaf of M. For how

many leaf nodes of M can the Challenge-Response be successful? You can assume that m, the number of leaves, is a power of 2. Explain your answer.

5. Assume the Master Hash is obtained by concatenating the values at the leaves (strings obtained by hashing the file blocks) and hashing the obtained concatenated string. What are the advantages and disadvantages of this approach? Explain your reasoning.

Grading

Project is 100 points. The bonus task is an additional 10 points.

- **30** Task 1: Building a Merkle tree (using MD5)
- **30** Task 2: Challenge and response
- **15** Task 3: Sum Hash
- **20** Report: Responses to experimental and analytical questions
- **5** Code Quality

Submission Instructions

You are responsible for ensuring your submission meets the requirements. If not, points will be deducted from code quality allocation. Your project will not be graded before the deadline. Your score on Vocareum will remain 0 until we grade your project.

Code

- Submit only the source files of your implementation. This means .java files only.
  - No .jar files or .class files should be submitted.
  - If a precompiled binary file is submitted without the .java file, you will receive a 0.
- In each of the `.java` files, you must correctly fill your name, login ID, your project completion date, and your PSO section number.
- Your code must compile to be graded by the autograder. Be sure to test your code before submitting. Do not change the folder structure. The code should compile with the given folder structure.
- You should submit a single folder named as “implementation” containing the following files (**ONLY** those files):
  - **Part 1:** MerkleTree.java
  - **Part 2**: Client.java and Server.java. Your report in one .pdf file. Maximum size allowed is 10MB.
  - **Optional Bonus**: CollisionGenerator.java

Report

- File name: `<username>.pdf`
- Submit your report with Part 2.
- Your report must be typed.
  - Use LaTeX, or OpenOffice Write, or Microsoft Word.
  - State your name on top of the report.
  - Handwritten reports, including digitized versions, will not be accepted.

  - Figures and diagrams that **cannot** be easily generated may be hand drawn and digitized for inclusion into the .pdf file representing your report. Note that tables and graphs that can easily be genrated by Excel or similar **cannot** be hand drawn!
- Your report must be submitted as a PDF.
  - Other file formats will not be accepted.
- Your report must be submitted using Vocareum.
  - Submissions via any other means, to any other venue, will not be accepted.

All work must be submitted in Vocareum following the submission instructions at Vocareum Submission Guidelines. Please be aware of:

- Only your final submission will be graded.
- A submission **after** the deadline is considered late. DON'T wait until the last second to submit.
- If you are submitting late and want to use your slip days, you need to select “use slip days” when submitting on Vocareum.
- See top of project description of allowed late/slip days for Part 1, Part 2, and Bonus Part.