

CmpE 160 - Introduction to Object Oriented Programming

Project #3 - Merkle Tree

Due Date: 02.06.2018 Thursday 23:55

In addition to its applications in the cryptography, file revision systems and most recently, the cryptocurrency technologies such as Bitcoin and Ethereum, Merkle Tree is a powerful mechanism that has been widely used in peer-to-peer file sharing, such as torrent protocol. For this project, you are expected to implement one of the most basic approaches to peer-to-peer (p2p) file sharing technology.

According to Wikipedia, p2p file sharing is the distribution and sharing of digital media using peer-to-peer networking technology. By using a p2p software, the users are able to share or get access to the content they desire. In this project, we limit our interest to the “downloading” part of the p2p file sharing. For this specific purpose, you need to study and understand how Merkle Trees work and then complete the unfinished project that you are provided to carry out the tasks listed below. Before elaborating more on your tasks, let us briefly describe the operation of the individual parts of the program that we expect you to develop in Section 1.

1. Overview

1.1 Merkle Tree:

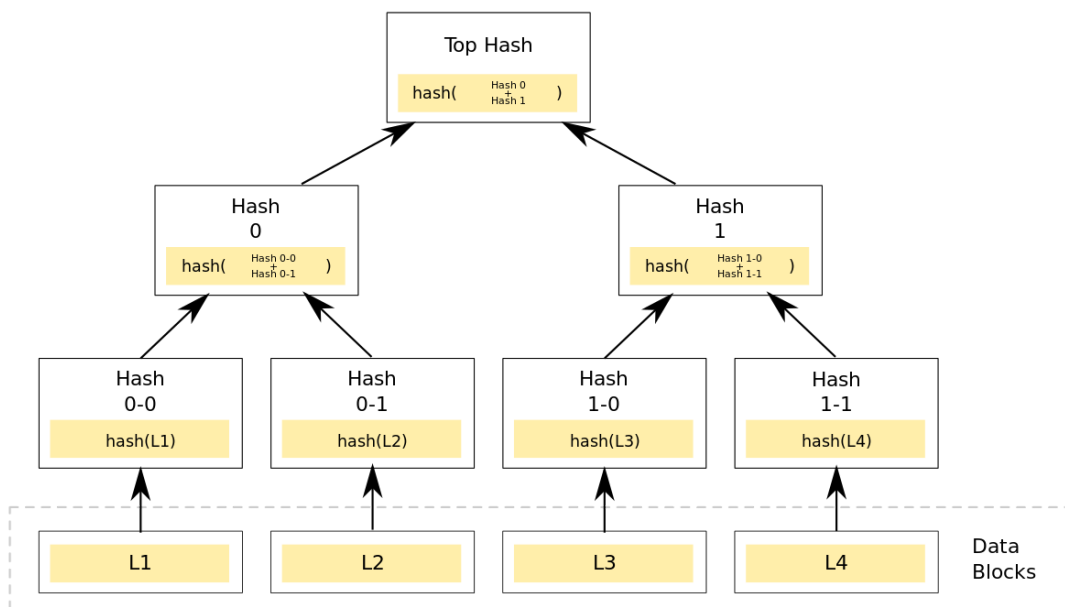


Figure 1: Merkle Tree, [1]

At the heart of this project lies the Merkle Tree. A **Merkle Tree (MT)** is a tree in which every leaf node is labelled with the **hash of a data block** and every non-leaf node is labelled with the **hash of the labels of its child nodes**, [1]. For example, in Figure 1, Hash0-0 node keeps the result of the hashing of the data labelled as L1 and Hash0 keeps the result of the hashing of the concatenation of the Hash0-0 and Hash0-1. That is,

$$\begin{aligned} \text{hash0-0} &= \text{hash}(L1) \\ \text{hash0} &= \text{hash}(\text{concat}(\text{hash0-0}, \text{hash0-1})) \end{aligned}$$

As mentioned above, there are many application areas for this specific data structure. In, for example, the original implementation of the Bitcoin, a version is implemented by Satoshi Nakamoto himself. This implies that there are a lot of versions of MTs but in the context of this project, we focus only on the plain, binary version.

In a binary MT, all of the **internal** nodes has at most 2 children, except the last two levels. The nodes in the penultimate level (the level above the last) contain only one child. As usual, the nodes in the ultimate layer are called **leaves**.

The advantages of using MTs in p2p file sharing systems are 2-fold:

1. By keeping the individual data chunks in the hard disk instead of RAM, it provides space efficiency. This is rather evident if we consider huge (>1GB) multimedia files.
2. Detection of the corrupt data chunk in *log* time.

The main objective of this project is to achieve these 2 benefits. Therefore in the following section, we try explain them in a realistic scenario.

A Simple P2P File Sharing Example

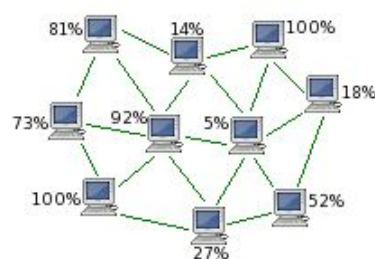
In the modern file sharing systems, in order to increase the speed of the download and decrease the risk of single-point failures, large files are divided into smaller data chunks and kept across different hosts.

Traditional Centralized Downloading



- Slow
- Single point of failure
- High bandwidth usage for server

Decentralized Peer-to-Peer Downloading



- Fast
- No single point of failure
- All downloaders are also uploaders

Figure 2: P2P System, [2]

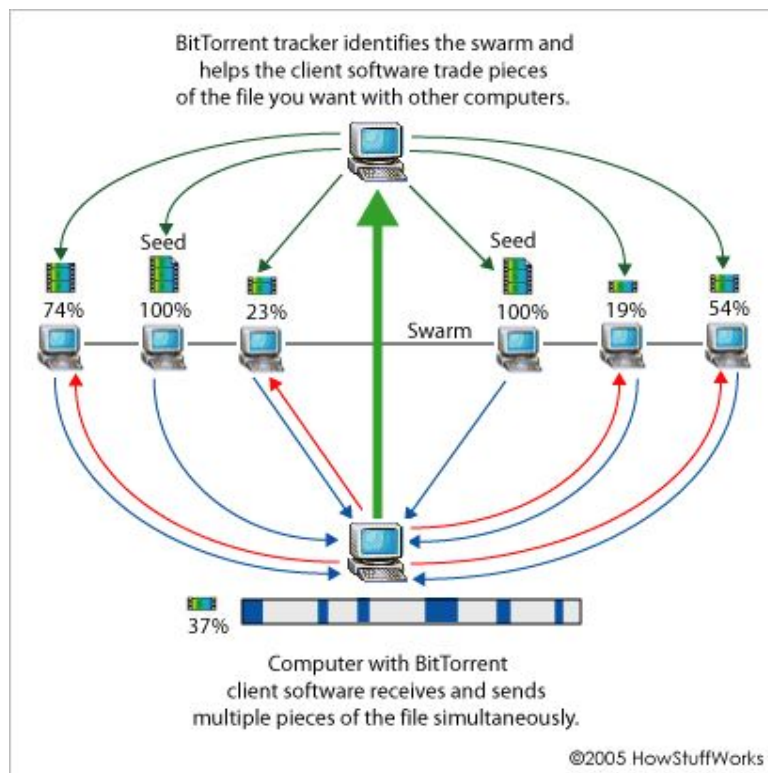


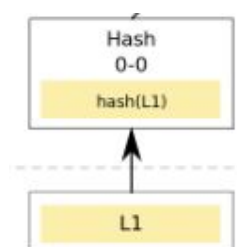
Figure 3: Bittorrent Example, [2]

When a user wants to download a file, she

1. Gets a list of the individual chunks from a **trusted** source.
2. Downloads the individual chunks from several sources (**uploaders**) asynchronously.

When the downloaded chunks are combined together, the user gets the file she wants. However the problem with this system is that, the uploaders are not to be trusted all of the time. Occasionally files get corrupt. Moreover, a malevolent software may easily be disguised as a data chunk.

MTs come in handy in these cases. In a simple case, just as in Figure 1, assume that the previously divided and distributed data contains 4 chunks. Assume further that we have already downloaded these chunks and created our MT as explained above.



- Hash0-0 keeps the result of the hashing of the first data chunk, Hash0-1 keeps the result of the hashing of the second and so on...
- In the upper level, Hash0 keeps the result of the hashing of the concatenation of Hash0-0 and Hash0-1.
- In the highest level is the root, keeps the result of the hashing of the concatenation of the Hash0 and Hash1

Evaluation of the integrity of the overall file is then just a **string comparison** of the hash coming from the trusted source and the one we just calculated and placed into the root node.

Whenever there is an inconsistency, since the error is propagated to the upper nodes, finding the corrupt chunk is quite easy as well. In a situation just as in Figure 4, we detect the inconsistency by comparing the root node. When we check its children in the next step, the right branch can be marked as valid, as a whole. In the next step we just focus on Hash0-0 and Hash0-1. This benefit is much more obvious in huge multimedia files that contain thousands of data chunks. This procedure saves a lot of time and resources.

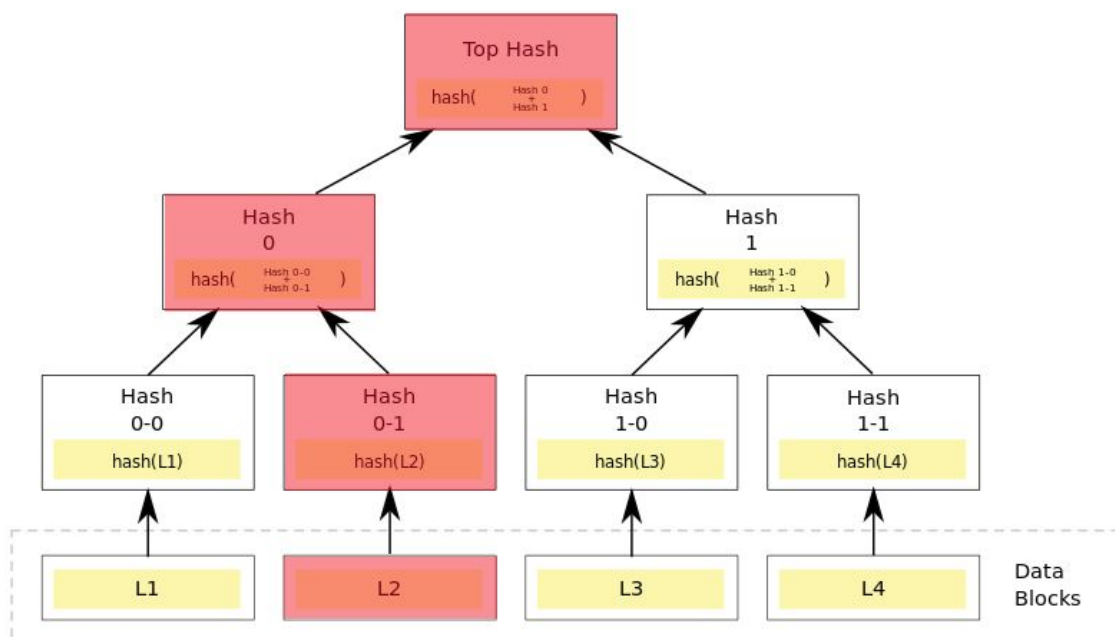


Figure 4: Propagation of the error

1.2 Hashing:

According to our definition at the lecture notes, hashing is a mapping between the search keys and indices. Since hashing is not the main focus of this project we are not going to go into the details.

In an MT implementation, we hash two types of data:

1. Files that each contain a data chunk
2. Strings that contain the concatenation of the hashes of the two children nodes.

Hashing is a widely applied procedure. Therefore, there are a lot of techniques in the literature. When the security of the data is not of vital importance, we may just use a checksum or CRCs as our hashing method. Although our case is such an example, we preferred to use a cryptographic hashing function **SHA-256**, that you are already given, to provide you with an easier use.

There is just a special case you need to pay attention. When the number of the nodes in a level is odd, as in Figure 5, at the end of the procedure of the formation of the upper-level nodes there would be one node left that you cannot pair. In order to calculate the hash of the its parent, just use it on its own. In the following example, there are 3 nodes in Level 2, so Hash0 is calculated regularly, whereas Hash1 is as follows:

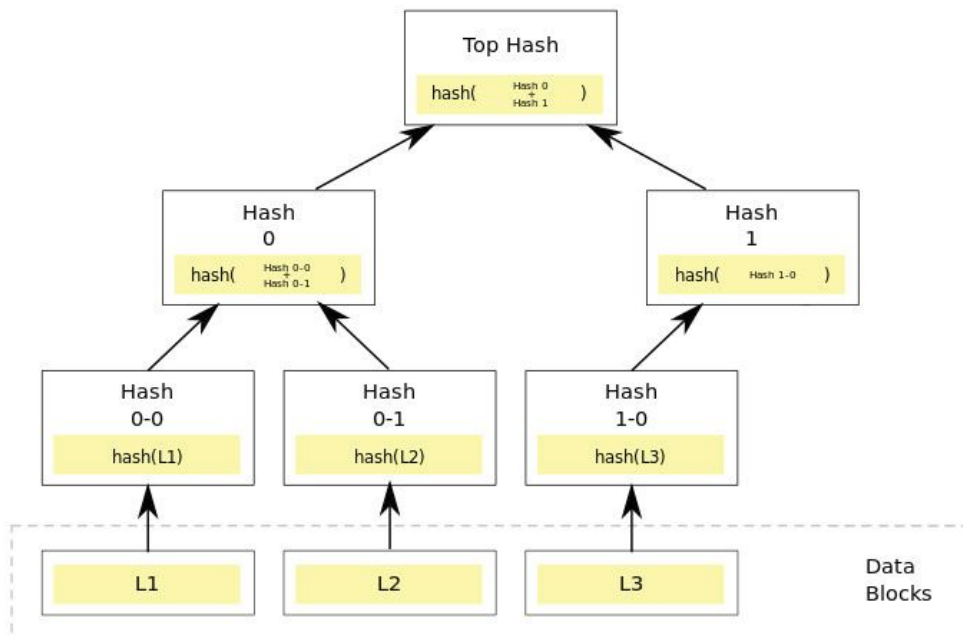
$$hash1 = hash(concat(hash1-0, ""))$$


Figure 5: Odd number of nodes

2. Contents

In your projects, you are given 3 packages. Do not let them limit you. You are free to implement your project however you like, the design is up to you. There of course are a couple of points that you **must** pay attention to. We mention them in the following. **Please read and understand the next three paragraphs**, otherwise you would probably get a compile error which leads to a grade of 0.

When grading your submissions, your version of *util* and *main* packages will be replaced by our version. Since you are supposed to use the *Main* class just for testing purposes, its deletion should be harmless. *HashGeneration* class will also be replaced for the sake of consistency. So, do not place your algorithm into these two packages.

Note that you are supposed to use the code you are given. The code in the *Main* class is there so that you do not make any mistakes when creating the given methods. Names and parameters of the given methods are of crucial importance. (As your codes will be automatically graded, class and method names are very important.) Furthermore, the code in the *HashGeneration* class is given so that we all use the same hash function. Use the methods in that class when you need to hash.

For example, currently these two lines are there in *Main.java*:

```
MerkleTree m0 = new MerkleTree("data/0.txt");
boolean valid = m0.checkAuthenticity("data/0meta.txt");
```

The first line tells you that you **need to** have a constructor that takes a String as its argument. The second line tells you that, this class **must** have a method called `checkAuthenticity` which also takes a `String` as its argument and returns an `boolean`. Also, the following example shows how you **need to** hash a `String`. These are mandatory, not optional.

```
HashGeneration.generateSHA256("winter is coming");
```

The data that you are going to use is a series of small images that are divided into 1 KB chunks. All is given in the *data* folder. In it, you will see a folder *split*, and many text files. There are 5 correct examples that you can test your code on, so 10 files can be used for testing purposes. Corresponding data files are in the *split* folder.

As can be understood just by reading the contents, with the two text files, you create and check your MT. Let us explain this on an example:

Example Case

There is a folder called *sample* in your projects. Inside, there is a folder named *split* that contains the folder of the data chunks, and the following files:

- *white_walker.jpg*: This is irrelevant. It is just the original version of the file before it is split into chunks.
- *white_walker.txt*: The file that contains the paths of the individual data chunks.
- *white_walkermeta.txt*: The file that contains the hash values of the nodes in Figure 6, in **Breadth-First-Traversal** order (There is an example of BFT in the lecture notes: Chapter 7, Tree Traversal, project in the last slide).

By reading from *white_walker.txt* file, you are supposed create the MT.

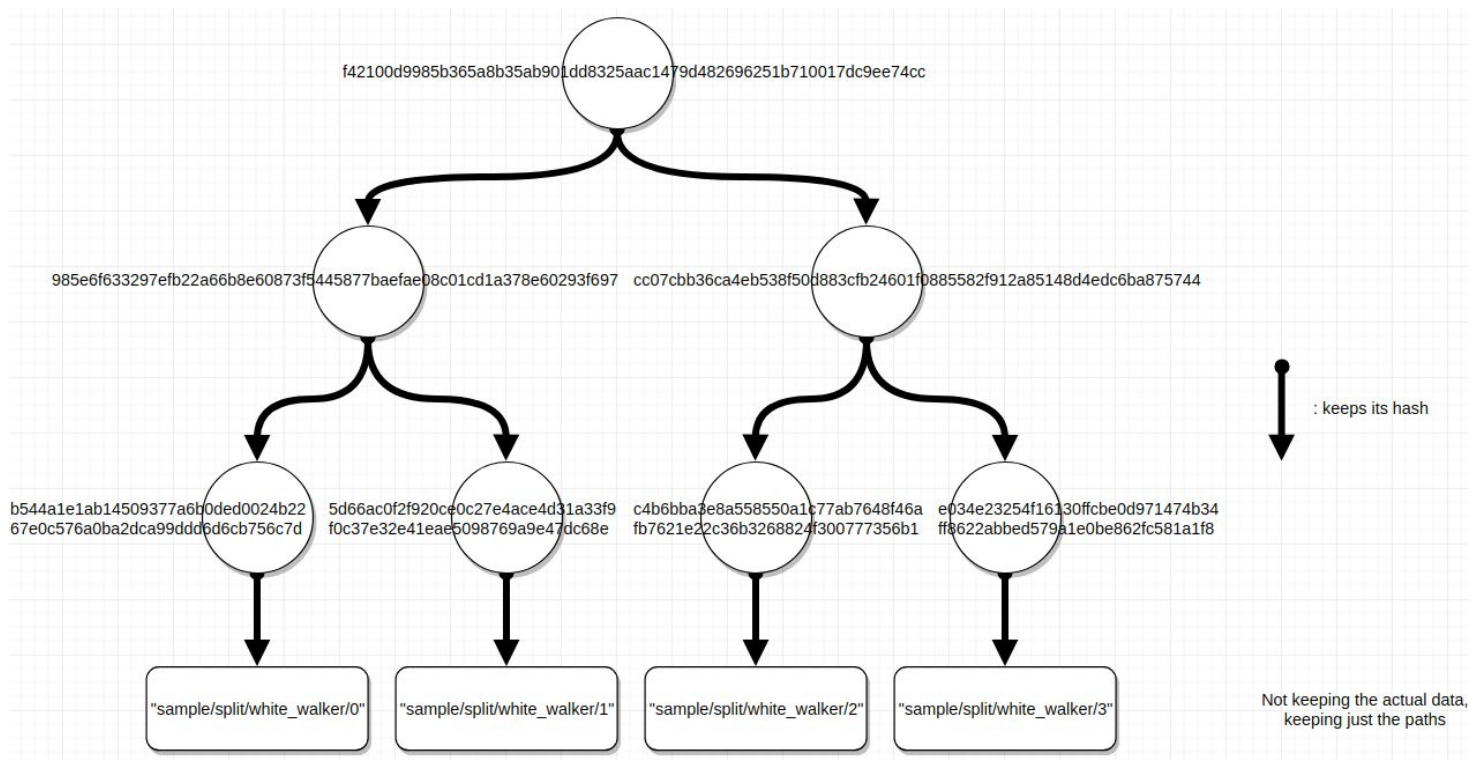


Figure 6: Sample MT

If you are able to create the MT, just as the one above, your next task is to compare -by starting from the root node- the equality of the hashes that you calculate with the hashes that your trusted source gave you. Use the meta file *white_walkermeta.txt* for this purpose.

If you notice any inconsistency, your next task is to find the source of it. There are additional examples in the data folder, for example *1_bad*, that contains at least one corrupt file. Its paths are given in the corresponding text file, the naming is self-explanatory. You can use the path files of the bad examples to form your tree and use regular meta files (e.g. *1meta.txt*) for the integrity check.

In the next section, we will one-by-one explain the requirements.

3. Requirements

There are two parts of this project: *primary* and *secondary*.

- In the *primary* part, you are supposed to complete the following tasks:
 1. Create a MT for a given configuration. (40 pts.): As explained in the example case above, assume that you have already
 - a. downloaded the chunks,
 - b. obtained the file that keeps the paths of the chunks,
 - c. meta file from the trusted source.

These are all given in the data folder. Use them to create the MT. There has to be a way for us to check the correctness of your MT. So, do have the required methods for the below functionalities:

1. `MerkleTree m = new MerkleTree(String path)`
2. `m.getRoot().getLeft().getRight().getData()`

2. Determine whether there are corruptions in the data chunks. (5 pts.): You need to be able to tell whether your download is fine or not. If there is not a corrupt file, your root hash should be equal to the first string in the meta file:

1. `boolean valid = m0.checkAuthenticity(String path);`

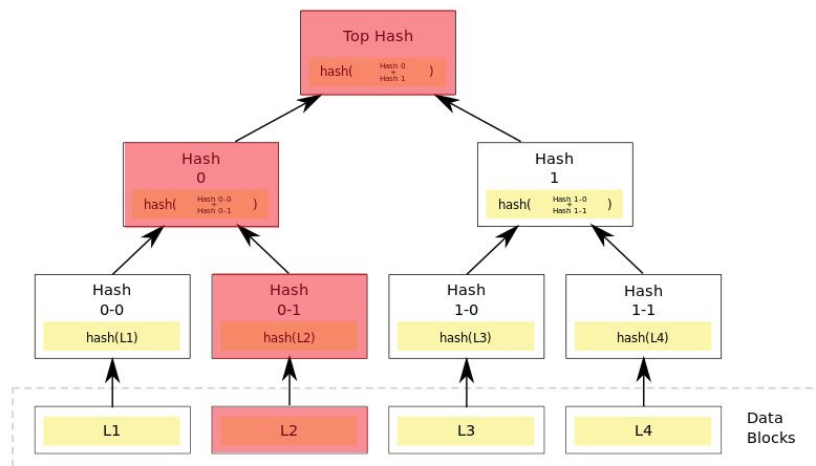
At this point, if you are confident enough in your design, try our hardest example; "9" in the data file. It is a larger file, so it needs much more calculations. Nevertheless, please beware that we might just use a larger file when we are grading your code.

3. If there are corruptions, find them exploiting the benefits of MTs. (30 pts.): The main advantage of using MT is to *prune* the search space. As mentioned in Section 1.1, for finding the corrupted data chunk, you do not need to traverse the subtree of a node if the hash of that node is correct. As an exemplary case of Figure 4, you do not traverse the right subtree of the root node since the corruption is resulted by the left subtree. Therefore, when one looks for the corrupted node, the subtrees of a node with the correct hash value can be pruned to shrink the search space.


For this purpose, you need to have another method. The signature of this method has to be as follows:

1. `public ArrayList<Stack<String>> findCorruptChunks(String path)`

As can be understood from the signature above, this method **must** return a Stack of Strings for each corrupt chunk in the data. Think about the case below:



Here, there is just one corrupt chunk, L2. Therefore your *findCorruptChunks* method must return an arraylist containing one stack with the hash values of the nodes as follows:

STACK	
	• 5d66ac0f2f920ce0c27e4ace4d31a33f9f0c37e32e41eae5098769a9e47dc68e (root->left->right)
	• 985e6f633297efb22a66b8e60873f5445877baefae08c01cd1a378e60293f697 (root->left)
	• f42100d9985b365a8b35ab901dd8325aac1479d482696251b710017dc9ee74cc (root)

In order to be able to achieve this, we advise you to use stacks and queues and **do not forget that** the meta list that you are given is in Breadth-First-Traversal order.

Suppose that there is just another corruption in the chunk L3. Then we would like the returned arraylist to keep that in its 2nd

- The *secondary* part evaluates your level of comprehension of Java, in general. In this part, we expect you to implement the remaining of the system to obtain the metadata from the trusted source and download the data chunks from several untrusted sources. (25 pts.)

Under *secondaryPart/data* folder there is a text file that contains URLs for 5 different examples. *robot.cmpe.boun.edu.tr* is the trusted source, where the meta data is kept. Chunks are kept in two different hosts and some of them are corrupt on purpose.

By using the data you download from the trusted source, you need to

1. Download the chunks from one source.
2. Create your MT.
3. Check the authenticity of the chunks.
4. If there is a problem, use the alternative source to overcome the problem.

This process should be initiated with a function call as follows:

```
download("secondaryPart/data/download_from_trusted.txt")
```

Keep the downloaded files under the *secondaryPart/data*.

4. Cheating

You are given a rather realistic project. You may think of copying pieces from online sources or others' codes. Consider yourselves warned that we plan to apply a very strict check for cheating.

You are free to find, adapt and use **code snippets, only if** you state the source, just as we did in the *HashGeneration* class. Any other similarity to any other source that we can find is highly likely to be regarded as cheating. (For example, copying an entire class from some page, using a method that you find somewhere, as it is etc...). Do not forget that sharing code with your classmates is strictly forbidden.

5. References

1. https://en.wikipedia.org/wiki/Merkle_tree
2. <https://www.quora.com/Why-do-torrents-both-download-and-upload-when-we-just-need-to-download-a-file-What-is-the-use-of-the-upload-function>
3. <https://blog.ethereum.org/2015/11/15/merkle-in-ethereum/>
4. <https://hackernoon.com/merkle-trees-181cb4bc30b4>

When we get ourselves familiar with the subject, we used the material in the above pages. The following are some videos that we find instructive:

1. <https://www.youtube.com/watch?v=WF5dNyFOqEc>
2. <https://www.youtube.com/watch?v=jKpkM7Zeb6U>
3. <https://www.youtube.com/watch?v=FDXPtOqtdE4>