CSCI 531 Programming Assignment 2 (100 points)

Note: this assignment is not as rigorous as usual CS projects. You should focus on the test cases listed in this document.

Your task is to implement a merkle hash tree and proofs of inclusion and consistency.

A merkle tree is a binary tree of hashes, in which all the leaf nodes are the individual data elements. To construct a merkle tree, the initial data elements are first hashed using the merkle tree hash function to generate the leaf nodes of the tree. The resulting hashed data are subsequently hashed together in pairs to create the parent nodes of the leaf nodes. This process continues until it results in a single hash known as the merkle root.

Project Details

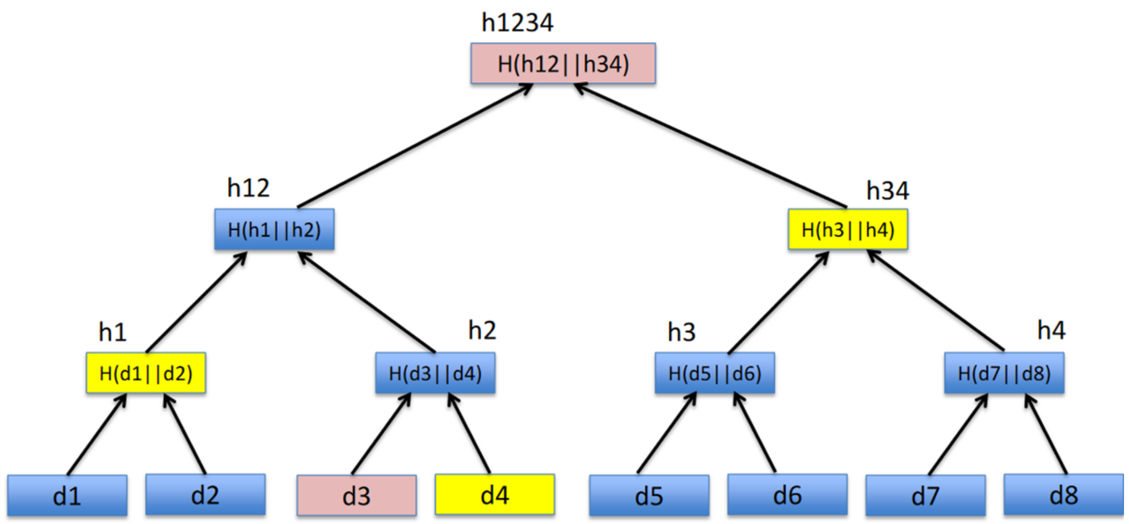
In this assignment you must write your own code (you *cannot* use any external code/library that implements merkle tree functionality). However, you can use built-in libraries to construct trees and calculate hash.

To complete the project, you will need to write two Python 3 programs:

1. **buildmtree.py** — generate merkle hash tree and print the content of the three into a file.
   - The program implements a merkle tree data structure, for representing a list of hashed data as a binary tree of pairs of hashes converging to a single hash, the merkle root. SHA_256 should be used as the hashing algorithm. For hash256, you can use a library. For example, 'from hashlib import sha256' and use it like this "hashlib.sha256(data.encode('utf-8')).hexdigest()"
   - The program takes a single command line argument: a list of strings out of which a merkle tree will be generated. For test purposes, use sting of names: **alice**, **bob, carlol, david**.
   - The program must be runnable directly from the command shell, e.g., ./buildmtree.py [alice, bob, carlol, david]. The input format can be slightly modified for better command line parsing.
   - The program must produce an output file called **merkle.tree** containing the printout of the built tree. You need to print the content of all nodes in a user-friendly way (clearly show the structure of the tree). The format of the output file is up to you, for example, you can use Json format.

2. **checkinclusion.py** proves that a certain string exists in the three.

   - This program takes single command line argument (a string) and returns yes/no answer along with a proof in case the answer is "yes".
   - For example, in the figure below given the root h1234 and the leaf d3 to authenticate, output 'yes' and the yellow nodes (d4, h1, h34) as proof.



- Time complexity is not the focus here as long as it's not ridiculous (e.g., over 30 minutes).
- The proof consists of the corresponding hashes that demonstrate that you can get from the list that contains the string in question to the tree root hash.
- For test purposes, use stings: **david** and **richard**
- The program must be runnable directly from the command shell, e.g., ./checkinclusion.py richard
- Then the following command would produce positive answer along with a proof indicated by three hashes:

```
./checkinclusion.py david


yes [d3, h1, h34]
```

3. **checkconsitency.py** proves that two versions of a data structure are consistent. That is: the new version includes everything in the old version in the same order.
   - The new version includes all the information in the old version
   - Everything is in the same order
   - All new records are placed after the older version

   You need to verify that the old merkle tree hash is a subset of the new merkle tree hash. Then you need to verify that the new merkle tree hash is the concatenation of the old merkle tree hash plus all the intermediate node hashes of the newly appended strings. The consistency proof is the minimum set of intermediate node hashes you need to compute these two things.
   Detail about verification algorithm: Section 2.1.3 of RFC 6962 https://tools.ietf.org/html/rfc6962

   - This program takes two command line arguments (two lists of strings) and returns yes/no answer along with a proof in case the answer is "yes".
   - The program generates an output file **merkle.trees** that contains printouts of the two merkle trees built from the two lists in a user-friendly format.

- For test purposes, use these lists of strings stings:

  [alice, bob, carlol, david] [alice, bob, carlol, david, eve, fred]

  [alice, bob, carlol, david] [alice, bob, david, eve, fred]

  [alice, bob, carlol, david] [alice, bob, carlol, eve, fred, david]

- Then the following command would produce positive answer along with a proof indicated by three hashes, where 45879 is the old root hash, 33890 is the new root hash:

  ```
  ./checkconsitency.py [alice, bob, carlol, david] [alice, bob, carlol, david, eve, fred]
  yes [45879, 23745, 33890]
  ```

- If the list is in order and the new string has even (2^n to be precise) number of nodes more than that of old string, then output yes. Then, in the example of [alice, bob, carlol, david] [alice, bob, carlol, david, eve, fred], output [oldRoot, intermideate node [Hash(eve|fred), newRoot].
  Then, in the example of [alice, bob, carlol, david] [alice, bob, carlol, david, eve, fred], output [oldRoot, intermideate node [H(eve|fred), newRoot]

In addition to the three Python programs, you must provide a written description of the design of your programs and a screen capture of a session demonstrating that your programs work.

The description of the design can also include a diagrammatic description of the simulation as shown in the following example: https://medium.com/ontologynetwork/everything-you-need-to-know-about-merkle-trees-82b47da0634a

For the test strings, show how the tree structure will change with each command.

For example, a screen capture of the following sequence of commands would be sufficient:

```
./buildmtree.py [alice, bob, carlol, david]

cat merkle.tree

./checkinclusion.py richard

no

./checkinclusion.py david

  yes [12345, 67890, 10111]

./checkconsitency.py [alice, bob, carlol, david] [alice, bob, carlol, david, eve, fred]

yes [45879, 23745, 33890]

cat merkle.trees

./checkconsitency.py [alice, bob, carlol, david] [alice, bob, david, eve, fred]

no
```

```
cat merkle.trees

./checkconsitency.py [alice, bob, carlol, david] [alice, bob, carol eve, fred, davis]

no

cat merkle.trees
```

Assignment Submission

Submit the assignment on DEN D2L. The submission will consist of three files:

1. A design document in PDF format providing a brief description of the design of your programs and including a screen capture of the working programs as described above.

2. The program buildmtree.py.

3. The program checkinclusion.py.

4. The program checkconsitency.py.

Grading

1. Design document (10 points)
2. Correct implementation of bildmtree.py (20 points)
3. Correct implementation of checkinclusion.py (30 points)
4. Correct implementation of checkconsitency.py (40 points)