Code Explanation for Merkle tree assignment
Jing Jiang

```python
import hashlib

class Merkle_tree:
    def __init__(self, transactions=None):
        hashed_t = []
        for t in transactions:
            hashed_t.append(hashlib.sha256(t.encode()).hexdigest())
        self.txs = hashed_t
        self.levels = None
```

1. Here I create a class called *Merkle_tree* and wrote its constructor. The parameter it takes is a list called transactions. Inside the constructor, I traversed the transactions and hash it into a new list called *hashed_t*. And then I created two properties: *self.txs* to store the hashed transactions and *self.levels* to store the tree data structure. The property *self.levels* is a two-dimensional list, say *levels[a][b]*. Index [a] indicates the level of the tree structure and index [b] indicates the nodes in a specific level. For example, the first one, *levels[0]* is the top level, which has only one hash: *levels[0][0]*, which is the root. And the second one, *levels[1]* has two values, which is the two children of *levels[0][0]*, denoted as *levels[1][0]* and *levels[1][1]*.

```python
    def generate_tree(self):
        if len(self.txs)>0:
            self.levels = [self.txs]
            while len(self.levels[0]) > 1:
                self.generate_next_level()


    def generate_next_level(self):
        odd_node = None
        level = []
        length = len(self.levels[0])

        if length%2!=0:
            odd_node = self.levels[0][-1]
            length = len(self.levels[0])-1
        for i in range(0,length-1,2):
            j = i + 1
            lr = self.levels[0][i] + self.levels[0][j]
            level.append(hashlib.sha256(lr.encode()).hexdigest())
        if odd_node is not None:
            level.append(hashlib.sha256(odd_node.encode()).hexdigest())
        self.levels = [level] + self.levels

    def get_root(self):
            if self.levels is not None:
                return self.levels[0][0]

            else:
                return None
```

2. These three methods in figure 2 allows me to generate the tree and get the root. In method *generate_next_level* I first detected whether there's odd number of data. If there is, store the rightmost transaction into a variable called *odd_node*., and the variable length, which is the length of the current top level, is minused by 1. Then I wrote a for loop steping by two, in each iteration the two nodes of a specific level was concated and hashed to form their parent node. And this parent node is appened to a list called *level*. If the *odd_node* is not None, which indicates there's odd number of nodes in this level, the *odd_node* itself got hashed and is appened to level. Finally, the list *level* was prepended to *self.levels* and become *levels[0]*, thus a new level is made. In method *generate_tree*, new levels are generated until the top level has a length of 1. In method *get_root*, the only node at the top level, which is the Merkle tree root, was returned.

```
def get_sibling_list(self, index):
    if self.levels is None or index > len(self.txs)-1 or index < 0:
        return None
    else:
        slist = []
        for x in range(len(self.levels)-1,0,-1):
            level_length = len(self.levels[x])
            if (index == level_length-1 and level_length%2 != 0):

                slist.append({"left": ''})
                index = int(index / 2)

                continue
            is_right_node = index % 2 != 0
            sibling_index = index - 1 if is_right_node else index + 1
            sibling_pos = "left" if is_right_node else "right"
            sibling_value = self.levels[x][sibling_index]
            slist.append({sibling_pos: sibling_value})
            index = int(index / 2)
        return slist

def proof_membership(self, index, target_hash):
    merkle_root = self.get_root()
    slist = self.get_sibling_list(index)
    if len(slist) == 0:
        return target_hash == merkle_root
    else:
        proof_hash = target_hash
        for proof in slist:
            try:
                sibling = proof['left']
                proof_hash = hashlib.sha256((sibling + proof_hash).encode()).hexdigest()
            except:
                sibling = proof['right']
                proof_hash = hashlib.sha256((proof_hash + sibling).encode()).hexdigest()
        if proof_hash == merkle_root:
            return "Membership verified"
        else:
            return "Non-membership"
```

3. In method *get_sibling_list*, I created a list called *slist* to store the sibling nodes at each level, given a certain index. If there's odd number of node, and the index points to the rightmost odd node, the sibling list will be concated with an empty string until it reached the second level, in which the odd node do have a sibling

node. The Boolean value *is_right_node* indicate whether this node is to the right of its sibling. If it is, a key *"left"* and the sibling hash value to its left will be appended to the *slist*; otherwise a key *"right"* and its right sibling hash value will be appended. This process goes from the bottom level to the top level and this method returns *slist*. A sibling list looks like this:

[{'left':
'ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb'},
{'right':
'd3a0f1c792ccf7f1708d5422696263e35755a86917ea76ef9242bd4a8cf4891a'},
{'right':
'd7aeed5b85cbff58f979412abf2af9b82dbbb56903f97071fd2d132e81bcf270'}]

In method *proof_membership*, an *index* and a *target_hash* is taken. Then the sibling list is generated and the *target_hash* and its sibling hash at each level are concated and hashed, all the way up to the top. And the hashed value is compared with the *merkle_root*. If it is the same, return the proof of membership; otherwise return the proof of non-membership.