TEST 2 - Hashed Datastructures

Q 1.

```
import hashlib

##############################hash pointers#########################

class HashPointer:
    def __init__ (self, target):
        self._target = target
        self._hash = self.digest()

    def verify(self):
        return self._hash == self.digest()

    def digest(self):
        m = hashlib.sha256(self._target)
        return m.hexdigest()


####################hashed linked list aka blockchain############

class Block:
    def __init__ (self, data, prev = None):
        self._prev = prev
        self._data = data
        self._prev_hash = prev.digest() if prev is not None else
bytearray(256)

    def digest(self):
        m = hashlib.sha256()
        m.update (self._prev_hash)
        m.update (self._data)
        return bytearray(m.hexdigest(), 'utf-8')

    def verify(self, root_hash):
        my_hash = self.digest()
        if (root_hash != my_hash):
            print ("Hash does not verify for block containing", self._data)
        return (root_hash == my_hash
                    and (not self._prev or self._prev.verify
(self._prev_hash)))

class Blockchain:
    def __init__(self):
        self._root_hash = bytearray (256)
        self._list = None
```

```
    def add_block (self, data):
        new_block = Block (data, self._list)
        self._root_hash = new_block.digest()
        self._list = new_block

    def verify (self):
        return not self._list or self._list.verify (self._root_hash)


data1 = bytearray ("Like a Virgin", 'utf-8')
data2 = bytearray ("True Blue", 'utf-8')

bc = Blockchain()
bc.add_block (data1)
bc.add_block (data2)

## the root hash of the blockchain
print(bc._root_hash)

## 78be111c45ee204554b663dcf507f31cd9c0c37c26da0ad9b67201960106642a
```

Q 2.

```
import hashlib

class MerkleLeaf:
    def __init__(self, data):
        self._data = data
        self._hash = hashlib.sha256 (data).hexdigest()

class MerkleNode:
    def __init__ (self, left, right):
        self._left = left
        self._right = right
        combined_hash = bytearray(left._hash + right._hash, 'utf-8')
        self._hash = hashlib.sha256 (combined_hash).hexdigest()
        # m = hashlib.sha256()
        # m.update (left._hash)
        # m.update (right._hash)
        # self._hash = bytearray(m.hexdigest(), 'utf-8')

def construct_Merkle (items):
    # create working list of leaf nodes
    work_list = list (map (MerkleLeaf, items))
    assert len (work_list) != 0
    while True:
        next_list = []
        for i in range (0, len (work_list) - 1, 2):
            next_list.append (MerkleNode (work_list[i], work_list[i+1]))
        if len (work_list) % 2 == 1:
```

```
            next_list.append (MerkleNode (work_list[-1], work_list[-1]))
        work_list = next_list
        if len (work_list) == 1:
            break
    return work_list[0]

items = ("aaa", "bbb", "ccc")
mt = construct_Merkle(items)

## the top hash of the Markle tree
print (mt)
print (mt._hash)
# <__main__.MerkleLeaf instance at 0x7ff819c2dc20>
# 56e962de2b5cdc0b8cd8d1929abfa96c831f64e0cf5ad23420ece8cb2ae77ddc

## the left hash of the Markle tree
print (mt._left)
print (mt._left._hash)
#<__main__.MerkleNode instance at 0x7fbe8badaab8>
#7607b2809ae92fffc220deb8af1e4c2878180c29e9f861d79efb0f8bb9961548

## the right hash of the Markle tree
print (mt._right)
print (mt._right._hash)
#<__main__.MerkleNode instance at 0x7fbe8badab48>
# 7d9bf113ceed7a50bacb7361ba2ac0f52f0a23f4d0357a6d69ba4d23cb0afb4a
```

Q 3.

```
import hashlib

class MerkleLeaf:
    def __init__(self, data):
        self._data = data
        self._hash = hashlib.sha256 (data).hexdigest()

class MerkleNode:
    def __init__ (self, left, right):
        self._left = left
        self._right = right
        combined_hash = bytearray(left._hash + right._hash, 'utf-8')
        self._hash = hashlib.sha256 (combined_hash).hexdigest()

def construct_Merkle (items):
    # create working list of leaf nodes
    work_list = list (map (MerkleLeaf, items))
    assert len (work_list) != 0
    while True:
        next_list = []
        for i in range (0, len (work_list) - 1, 2):
            next_list.append (MerkleNode (work_list[i], work_list[i+1]))
```

```
        if len (work_list) % 2 == 1:
            next_list.append (MerkleNode (work_list[-1], work_list[-1]))
        work_list = next_list
        if len (work_list) == 1:
            break
    return work_list[0]

# Proof of Membership in a Merkle Tree
def verify(top_hash, data_hash, proof):
    assert len (proof) != 0
    tmp = data_hash
    for i in range (0, len (proof), 1):
        tmp = hashlib.sha256 (bytearray(tmp + proof[i][0], 'utf-
8')).hexdigest() if proof[i][1] == "Right" else  hashlib.sha256
(bytearray(proof[i][0] + tmp, 'utf-8')).hexdigest()
    return (tmp == top_hash)


vef =
verify('56e962de2b5cdc0b8cd8d1929abfa96c831f64e0cf5ad23420ece8cb2ae77ddc',
'3e744b9dc39389baf0c5a0660589b8402f3dbb49b89b3e75f2c9355852a3c677',
[('9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0',
"Left"),
('7d9bf113ceed7a50bacb7361ba2ac0f52f0a23f4d0357a6d69ba4d23cb0afb4a',
"Right")])

print (vef)
# True
```

Q 4.

To prove that an element x, which is smaller than any element in a sorted Merkle tree, is not an element of the tree it is sufficient to give a path to the median element and show that this path is a rightmost path.

Q 5.

```
import hashlib
from array import *

class MerkleLeaf:
    def __init__(self, data):
        self._data = data
        self._hash = hashlib.sha256 (data).hexdigest()

class MerkleNode:
    def __init__ (self, left, right):
        self._left = left
        self._right = right
```

```
        combined_hash = bytearray(left._hash + right._hash, 'utf-8')
        self._hash = hashlib.sha256 (combined_hash).hexdigest()

def construct_Merkle (items):
    # create working list of leaf nodes
    work_list = list (map (MerkleLeaf, items))
    assert len (work_list) != 0
    while True:
        next_list = []
        for i in range (0, len (work_list) - 1, 2):
            next_list.append (MerkleNode (work_list[i], work_list[i+1]))
        if len (work_list) % 2 == 1:
            next_list.append (MerkleNode (work_list[-1], work_list[-1]))
        work_list = next_list
        if len (work_list) == 1:
            break
    return work_list[0]


# check for right most
def rightmost(proof):
    for e in proof:
      if e[1] == "Right":
            return False
    return True

# check for left most
def leftmost(proof):
    i = 0
    while i < len (proof):
      if proof[i][1] == "Left":
            return False
      i += 1
    return True

# Direct Neighbors
def check_DirectNeighbors(proof1, proof2):
    if len (proof1) == 0 or len (proof2) == 0:
      return False
    if proof1[-1][0] == proof2[-1][0]:
      if proof1[-1][1] == proof2[-1][1]:
            return check_DirectNeighbors(proof1[:-1], proof2[:-1])
    if proof1[-1][1] == "Left" and proof2[-1][1] == "Right":
      return leftmost(proof1[:-1]) and rightmost(proof2[:-1])
    if proof1[-1][1] == "Right" and proof2[-1][1] == "Left":
      return rightmost(proof1[:-1]) and leftmost(proof2[:-1])
    return False
```

```
proof1 =
[('9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0',
"Left"),
('7d9bf113ceed7a50bacb7361ba2ac0f52f0a23f4d0357a6d69ba4d23cb0afb4a',
"Right")]
#bbb : aaa cccccc
proof2 =
[('3e744b9dc39389baf0c5a0660589b8402f3dbb49b89b3e75f2c9355852a3c677',
"Right"),
('7d9bf113ceed7a50bacb7361ba2ac0f52f0a23f4d0357a6d69ba4d23cb0afb4a',
"Right")]
#aaa : bbb cccccc
print check_DirectNeighbors(proof1, proof2)
# True

Q 6.
import hashlib
from array import *

class MerkleLeaf:
    def __init__(self, data):
        self._data = data
        self._hash = hashlib.sha256 (data).hexdigest()

class MerkleNode:
    def __init__ (self, left, right):
        self._left = left
        self._right = right
        combined_hash = bytearray(left._hash + right._hash, 'utf-8')
        self._hash = hashlib.sha256 (combined_hash).hexdigest()

def construct_Merkle (items):
    # create working list of leaf nodes
    work_list = list (map (MerkleLeaf, items))
    assert len (work_list) != 0
    while True:
        next_list = []
        for i in range (0, len (work_list) - 1, 2):
            next_list.append (MerkleNode (work_list[i], work_list[i+1]))
        if len (work_list) % 2 == 1:
            next_list.append (MerkleNode (work_list[-1], work_list[-1]))
        work_list = next_list
        if len (work_list) == 1:
            break
    return work_list[0]

def verify(top_hash, data_hash, proof):
    assert len (proof) != 0
    tmp = data_hash
    i = 0
```

```
    while True:
        tmp = hashlib.sha256 (bytearray(tmp + proof[i][0], 'utf-
8')).hexdigest() if proof[i][1] == "Right" else  hashlib.sha256
(bytearray(proof[i][0] + tmp, 'utf-8')).hexdigest()
      i = i + 1
      if i == len (proof):
            break
    return (tmp == top_hash)


# check for right most
def rightmost(proof):
    for e in proof:
      if e[1] == "Right":
            return False
    return True

# check for left most
def leftmost(proof):
    i = 0
    while i < len (proof):
      if proof[i][1] == "Left":
            return False
      i += 1
    return True

# Direct Neighbors
def check_DirectNeighbors(proof1, proof2):
    if len (proof1) == 0 or len (proof2) == 0:
      return False
    if proof1[-1][0] == proof2[-1][0]:
      if proof1[-1][1] == proof2[-1][1]:
            return check_DirectNeighbors(proof1[:-1], proof2[:-1])
    if proof1[-1][1] == "Left" and proof2[-1][1] == "Right":
      return leftmost(proof1[:-1]) and rightmost(proof2[:-1])
    if proof1[-1][1] == "Right" and proof2[-1][1] == "Left":
      return rightmost(proof1[:-1]) and leftmost(proof2[:-1])
    return False

# Non Occur
def check_Nonoccur(x, top_hash, proof):
    assert len (proof) != 0

    if len(proof) == 1:
       return (x < proof[0][0] and verify(top_hash,  hashlib.sha256
(proof[0][0]).hexdigest(), proof[0][1]) and leftmost(proof[0][1])) or (x >
proof[0][0] and verify(top_hash, hashlib.sha256 (proof[0][0]).hexdigest(),
proof[0][1]) and rightmost(proof[0][1]))
    if len(proof) == 2:
```

```
      return proof[0][0] < x and x < proof[1][0] and verify(top_hash,
hashlib.sha256 (proof[0][0]).hexdigest(), proof[0][1]) and verify(top_hash,
hashlib.sha256 (proof[1][0]).hexdigest(), proof[1][1]) and
check_DirectNeighbors(proof[0][1], proof[1][1])

    return False

#Tests

items = ("2","4","6","8")
top_hash = construct_Merkle(items)._hash

proof1 = [("2",
[("4b227777d4dd1fc61c6f884f48641d02b4d121d3fd328cb08b5531fcacdabf8a",
"Right"),
("1be9de78090d759848611d01dda5ceee9c07c1b7246682a4d99c56bfa58d36ae",
"Right")]), ("4",
[("d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35",
"Left"),
("1be9de78090d759848611d01dda5ceee9c07c1b7246682a4d99c56bfa58d36ae",
"Right")])]

proof2 = [("2",
[("4b227777d4dd1fc61c6f884f48641d02b4d121d3fd328cb08b5531fcacdabf8a",
"Right"),
("1be9de78090d759848611d01dda5ceee9c07c1b7246682a4d99c56bfa58d36ae",
"Right")])]

proof3 = [("6",
[("2c624232cdd221771294dfbb310aca000a0df6ac8b66b696d90ef06fdefb64a3",
"Right"),
("de777286dda88425ee22c5c67a3f3725f5d29ff47248974aa47f46dd4f605681",
"Left")]), ("8",
[("e7f6c011776e8db7cd330b54174fd76f7d0216b612387a5ffcfb81e6f0919683",
"Left"),
("de777286dda88425ee22c5c67a3f3725f5d29ff47248974aa47f46dd4f605681",
"Left")])]

proof4 = [("8",
[("e7f6c011776e8db7cd330b54174fd76f7d0216b612387a5ffcfb81e6f0919683",
"Left"),
("de777286dda88425ee22c5c67a3f3725f5d29ff47248974aa47f46dd4f605681",
"Left")])]
```