## Exercise 1 (Working with Merkle Trees)

In the following, consider a binary merkle tree which is given by its list of leaves. In particular, we assume for simplicity that the length of the given list is a power of two.

1. Write a function which computes the merkle root.

code/merkleroot.py

```
import hashlib
import binascii


def hash(x):
    return hashlib.sha256(str(x).encode()).digest()


def merkle_root(datalist):
    """Returns the merkle root of a merkle tree defined by
    the leaves in datalist"""
    hashlist = [hash(x) for x in datalist]
    while len(hashlist) > 1:
        pairs = zip(hashlist[::2], hashlist[1::2])
        hashlist = [hash(x[0] + x[1]) for x in pairs]
    return hashlist[0]


thedatalist = ['I', 'am', 'Satoshi', 'Nakamoto']

if __name__ == "__main__":
    mr = merkle_root(thedatalist)
    print(binascii.hexlify(mr))
```

2. Write a function which creates the merkle proof for a given leaf.

3. Write a function which verifies a merkle proof.

code/treeaddr.py

```python
# tree addressing functions

from math import log


def parent(node):
    return (node - 1) // 2

def is_root(node):
    return node == 0

def is_leaf(node, tree_height):
    return node >= 2 ** tree_height - 1

def leaf_of_index(i, tree_height):
    return 2 ** tree_height - 1 + i

def index_of_leaf(node, tree_height):
    return node - 2 ** tree_height + 1

def sibling(node):
    if node % 2 == 0:
        return node - 1
    else:
        return node + 1

def child1(node):
    return 2 * node + 1

def child2(node):
    return 2 * node + 2

def tree_height(length):
    return int(log(length, 2))
```

code/merkleproof.py

```python
import binascii
from treeaddr import *
from merkleroot import hash, thedatalist


def node_hash(node, datalist):
    """Returns the hash value of a node in a merkle tree given by
    the leaves in datalist"""
    h = tree_height(len(datalist))
    if is_leaf(node, h):
        index = index_of_leaf(node, h)
        return hash(datalist[index])
    else:
        return hash(node_hash(child1(node), datalist) +
                    node_hash(child2(node), datalist))


def merkle_proof(node, datalist):
    """Returns the merkle proof of node in a merkle tree given by
    the leaves in datalist"""
    if is_root(node):
        return []
    s = sibling(node)
    return [(s, node_hash(s, datalist))] + merkle_proof(parent(node), datalist)


def merkle_root(thehash, proof):
    """Returns the merkle root computed from a hash and a merkle proof"""
    if proof == []:
        return thehash
    node, nodehash, tail = proof[0][0], proof[0][1], proof[1:]
    if node % 2 == 0:
        return merkle_root(hash(thehash + nodehash), tail)
    else:
        return merkle_root(hash(nodehash + thehash), tail)


p = merkle_proof(4, thedatalist)
print(p)

mr = node_hash(0, thedatalist)
print(binascii.hexlify(mr))

mr2 = merkle_root(hash('am'), p)
print(binascii.hexlify(mr2))
```

## Exercise 2 (Using Merkle Trees for Our File Storage Provider)

Note how in the file storage protocol of the last exercise sheet the client-side storage grows linearly in the number of files.

Design a similar protocol where the client only stores a single hash (a merkle root) and where the time for verification is logarithmic in the number of files (verification of a merkle proof). For simplicity, don't handle errors and don't optimize for performance.

1. Start with a protocol where the client can only read (so the server only has a `read` function).

**Implementation Client**

Client stores:

- a list of the names: `names`
- and one hash: `the_merkle_root`

```
def read(name):
    content = server.read(name)
    proof = server.get_merkle_proof(name)
    if the_merkle_root != merkle_root(hash(content), proof):
        throw error("server lied!")
    return content
```

**Implementation Server**

Server stores:

- dictionary `contents`
  key: filename, value: file content

```
def read(name):
   return contents[name]

def get_merkle_proof(name):
   i = indexof(name, sort(keys(contents)))
   node = leaf_of_index(i, tree_height(len(contents)))
   return merkle_proof(node, contents.values())
```

2. Add the `update` function.

**Implementation Client**

```
def update(name, content):
   proof = server.get_merkle_proof(name)
   thehash = server.get_hash(name)
   if the_merkle_root != merkle_root(thehash, proof):
      throw error("server lied!")
   the_merkle_root = merkle_root(hash(content), proof)
   server.update(name, content)
```

**Implementation Server**

```
def update(name, content):
   contents[name] = content

def get_hash(name):
   return hash(contents[name])
```

3. Add the `create` function (`delete` is similar).

**Implementation Client**

```
def create(name, content):
    leaves = server.get_leaf_hashes()
    if the_merkle_root != merkle_root(leaves):
        throw error("server lied!")
    names.add(name)
    names.sort()
    index = names.index(name)
    leaves.insert(index, hash(content))
    the_merkle_root = merkle_root(leaves)
    server.create(name, content)
```

**Implementation Server**

```
def create(name, content):
    contents[name] = content

def get_leaf_hashes():
    names_sorted = sort(keys(contents))
    f = lambda name: hash(contents[name])
    leaf_hashes = map(f, names_sorted)
    return leaf_hashes
```