2022암호경진대회 2번 문제

1. 아래 그림과 조건들을 만족하는 블록체인 모델을 구현하시오.

1.1 Block은 블록체인의 각 블록 구현을 위한 클래스입니다. 이 클래스에는 str 객체인 block_id, hash_previous_block과, list 객체인 block_data가 있습니다.

1.2 이 클래스에는 블록 초기화를 위한 여러 메소드가 내장되어 있습니다.

padding(trans, num)는 (str) trans가 (int) num에 맞도록 왼쪽에 0으로 채우거나, 왼쪽의 값을 잘라냅니다. 이때 매개변수 num은 자릿수를 의미합니다.

```
def padding(trans, num):
    trans_str = str(trans)

    if len(trans_str) < num: #trans의 자릿수가 num 보다 작은 경우
        #모자란 자릿수만큼 앞에 0을 붙여줍니다.
        trans_str_padded = '0' * (num - len(trans_str)) + trans_str
    else: #trans의 자릿수가 num 보다 같거나 큰 경우
        #num의 길이에 맞춰 앞부분을 잘라내어 줍니다.
        trans_str_padded = trans_str[-1 * num:]

    #자릿수를 맞춰준 trans를 반환합니다.
    return trans_str_padded
```

코드1. padding(trans, num)

transaction()는 난수를 생성하고, “블록 데이터를 구성하는 각 Transaction은 160 비트의 TxID값과 (편의상random 하게 생성된)864비트 크기를 갖는 Transaction값으로 구성됨” 이라는 조건에 맞도록 108자리로 패딩합니다.

```
def transaction():
    #난수를 생성합니다. 난수 생성범위는 임의로 설정하였습니다.
```

```
    trans = random.randrange(1, 10 * 99)
    #각 transaction은 864비트 크기의 transaction 값을 가집니다. 문자 하나가 8비트이므로 108자리로 padding 해줍니다.
    trans = Block.padding(trans, 108)

    return trans
```

코드2. transaction()

make_hash(test)는 (str) test를 SHA3-256을 이용하여 암호화하고 16진수로 문자화한 후, 하위 20글자를 돌려줍니다.

```
def make_hash(test):
    #hashlib.sha3_256()으로 객체 s를 생성합니다.
    s = hashlib.sha3_256()
    #update() 함수를 호출하여 utf-8 형식의 바이트 문자열 test를 해싱합니다.
    s.update(bytes(test, 'utf-8'))
    #hexdigest() 함수를 이용하여 바이트를 16진수로 변환한 문자열을 반환합니다.
    full_hash = s.hexdigest()
    #하위 20자리만 반환합니다.
    new_hash = full_hash[-20:]
    #가독성 제공을 위해 Base-58로 인코딩합니다.
    new_hash = str(base58.b58encode(new_hash))[2:-1]

    return new_hash
```

코드3. make_hash(test)

set_block_id(self)는 block_id가 160비트, 혹은 20바이트가 되도록 padding(block_id_counter, 20)을 사용하여 값을 정하고, Base-58로 인코딩합니다.

```
def set_block_id(self):
    global block_id_counter

    #blockID가 20바이트가 되도록 padding 해줍니다.
    self.block_id = self.padding(block_id_counter, 20)
    #padding한 blockID를 Base-58로 인코딩합니다.
    self.block_id = str(base58.b58encode(self.block_id))[2:-1]
```

코드4. set_block_id(self)

set_block_data(self)는 block data의 값을 block_data에 저장합니다. block_data는 8192개의 str을 담고 있는 list입니다. 각 str은 160 bits (20 bytes)의 TxID와 864 bits (108 bytes)의 transaction으로 이루어져 있습니다. 따라서 각 str은 128 bytes라고 할 수 있습니다. 이때, 전체 block data가 1MB이어야 한다는 점을 고려하면, 1MB / 128 byes = 1K * 1K bytes / 128 bytes = 2 ^ 10 * 2 ^ 10 bytes / 2 ^ 7 bytes = 2 ^ 13 = 8192 개의 TxID-transaction str 쌍이 block data에 존재해야 함을 알 수 있습니다. 각 str은 생성된 후 Base-58로 인코딩되고, block_data에 저장됩니다.

```
def set_block_data(self):
    #2^13(= 8192)개의 TxID-transaction str 쌍이 존재해야 합니다.
    for i in range(2 ** 13):
        #TxID는 1부터 시작하고 20바이트(= 160비트)이어야 합니다.
        tx_id = self.padding(i + 1, 20)
        trans = self.transaction()

        #Base-58로 인코딩합니다.
        temp = str(base58.b58encode(tx_id + trans))[2:-1]

        #block_data에 저장합니다.
        self.block_data.append(temp)
```

코드4. set_block_data(self)

set_merkle_tree(self)는 특정 블록이 변경되었는지 여부를 빠르게 확인할 때 유용한 merkle tree를 만드는 함수입니다. 해당 설명은 아래에서 하겠습니다.

set_hash_value(self)는 이전 블록의 block_id, hash_previous_block(이전 해시 값을 담아놓은 변수), 그리고 TxID-transaction 쌍들을 모두 해싱한 최종값을 통해서 다음 블록의 hash_previous_block을 만드는 메소드입니다.

```
def set_hash_value(self):
    global block_id_counter
    global blockchain
    global merkle_tree


    if block_id_counter == 0: #Genesis 블록의 blockID는 0x0이기 때문에 이전 해시 값은 0을 해싱한 결과입니다.
        self.hash_previous_block = self.make_hash('0' * 20)
    else: #이후 블록들의 hash_previous_block(이전 해시 값)은 이전 블록의 blockID와 해시 값, blockData를 더해서 해싱합니다.
        self.hash_previous_block ₩
            = self.make_hash(blockchain[block_id_counter - 1].block_id
                             + blockchain[block_id_counter - 1].hash_previous_block
                             + merkle_tree[block_id_counter - 1][13][0])
```

코드5. set_hash_value(self)

initialize_block(self)는 set_block_id(), set_block_data(), set_merkle_tree(), set_hash_value()를 차례로 실행하여 하나의 블록을 만들어냅니다. 그 이후 전역 변수인 block_id_counter을 1만큼 증가시킵니다.

```
def initialize_block(self):
```

```
    global block_id_counter



    self.set_block_id()

    self.set_block_data()

    self.set_merkle_tree()

    self.set_hash_value()



    block_id_counter += 1
```

코드6. initialize_block(self)

2. 개발한 블록체인의 구성을 고속화하기 위해 해시함수를 고속화하거나 고속 해시 값 검증 구조를 제시하고 이를 구현하시오.

개발한 블록체인의 구성을 고속화하기 위해 Block Data 의 해시 값을 생성할 때 머클 트리(Merkle Tree)를 사용하였습니다. 블록체인을 생성한 이후에 특정 transaction 값이 위변조 되었는지 빠르게 검증하기 위해 블록 하나 당 머클 트리를 하나씩 만들었습니다. 먼저, SHA3-256 해시함수를 사용하여 각 transaction 의 해시 값을 생성합니다. 그리고 순차적으로 가장 가까운 값 2 개씩 이어 붙인 뒤 다시 해시 값을 생성합니다. 이 과정을 마지막 하나의 값만이 남을 때까지 반복합니다.

```
def set_merkle_tree(self):
    global merkle_tree
    global index_merkle_tree
    num_iter = 2 ** 13
    row = 0

    #merkle tree 에 빈 리스트를 추가해서 TxID + Transaction 을 해싱한 값을 저장할 공간을 만듭니다. 모든 블록의 blockData 생성 과정을 담을 리스트입니다.
    merkle_tree.append([])

    while num_iter >= 1:
        #해당 인덱스에 리스트를 추가합니다. 아래 그림에 있는 Group 들이 됩니다.
        merkle_tree[index_merkle_tree].append([])
        if row == 0:
            for i in range(num_iter):
                merkle_tree[index_merkle_tree][row].append₩
                    (self.make_hash(self.block_data_list_id_trans[i]))
        else:
            for i in range(num_iter):
                merkle_tree[index_merkle_tree][row].append₩
```

```
                (self.make_hash(merkle_tree[index_merkle_tree][row - 1][2 * i]
                                + merkle_tree[index_merkle_tree][row - 1][2 * i + 1]))

        num_iter //= 2
        row += 1

    index_merkle_tree += 1
```

코드 7. set_merkle_tree(self)

transaction 각각의 해시 값을 하나의 리스트에 저장하고 그 리스트를 0 그룹이라고 가정합니다. 0 그룹의 요소들로 다시 해시 값을 생성하고 새로운 리스트에 저장합니다. 이 그룹은 1 그룹이라고 가정하면, 이렇게 생성된 그룹들(리스트)을 하나의 리스트로 또 생성하는 것입니다. 따라서 2 차원 리스트를 생성하게 됩니다.

앞서 설명한 바와 같이 Block Data 의 해시 값을 생성하였기 때문에, 특정 transaction 값이 위변조 되었음을 확인하기 위해 모든 해시 값을 알 필요가 없습니다. 그룹 당 하나의 해시 값만 알면 됩니다. 0 그룹에서는 검증하고자 하는 transaction 의 해시 값과 가장 가까운 노드의 값을 알면 됩니다. 1 그룹부터는 이전 그룹에서 선택한 노드의 부모 노드와 가장 가까운 노드의 값을 알면 됩니다. 예를 들어, 8 개의 transaction 들 중 첫 번째 transaction 이 위변조 되었는지 알고 싶다고 가정해봅시다. 검증하고자 하는 transaction 의 해시 값을 제외하고는 각 그룹 당 하나의 해시 값만 알면 되기 때문에 그림과 같은 노드의 값들만 알아내면 됩니다.

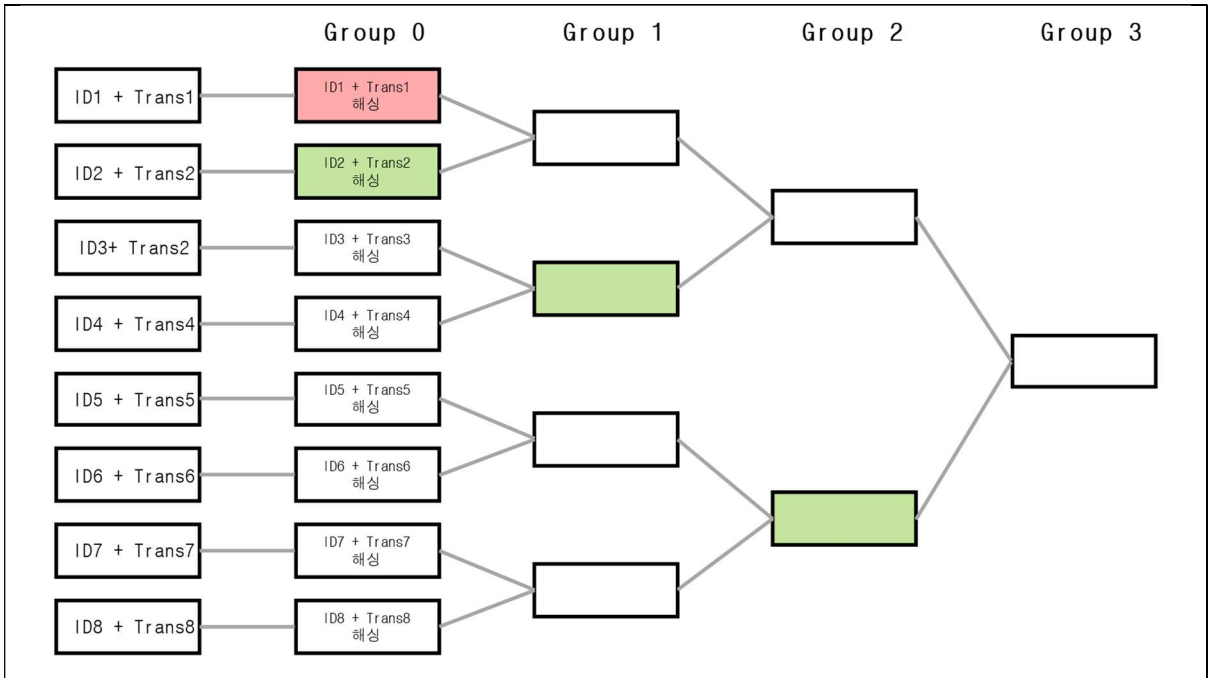

그림 1. merkle tree 예시

노드의 값을 2 차원 리스트에 저장했기 때문에 그룹명이 세로 인덱스가 되고, 그룹 내에서의 순서가 가로 인덱스가 됩니다. 우선, 검증할 노드에서 시작합니다. 노드의 가로 인덱스를 2 로 나누었을 때 나머지가 0 이면 다음에 찾을 노드의 가로 인덱스는 현재 노드의 가로 인덱스에 1 을 더한 값입니다. 반면에 나머지가 1 이면 현재 노드의 가로 인덱스 보다 1 작은 값이 됩니다. 또한, 찾은 노드의 부모 노드 좌표를 알려주어야 합니다. 부모 노드의 세로 인덱스는 현재 노드의 세로 인덱스 보다 1 큰 값이 되고, 가로 인덱스는 현재 노드의 가로 인덱스를 2 로 나눈 몫이 됩니다.

```
def find_address(tx_id):
    # 마지막에 더해야하는 좌표들 리스트
    lt2 = []

    if tx_id % 2 == 0:
        tx_id_result = tx_id + 1
        lt2.append(tx_id_result)
    else:
        tx_id_result = tx_id - 1
        lt2.append(tx_id_result)

    tx_id_next = tx_id // 2
    lt2.append(tx_id_next)

    return lt2
```

코드 8. 노드 찾기

3. 구현한 블록체인 모델을 사용하여, 특정 블록의 Transaction 데이터가 위변조 되었을 때, 해당 블록이 변경되었음을 보이시오.

change_block(id_block, tx_id)

1. tx_id -= 1

   TxID는 1부터 시작하는데 block_data는 list라서 index가 0부터 시작하기 때문에 일단 1을 빼준다.

2. temp = blockchain[id_block].block_data[tx_id][20:]

   blockchain[id_block].block_data[0:19]는 패드된 TxID값이다.

3. 108자리로 패드된 난수가 blockchain[id_block].block_data[tx_id][20:]과 다를 때까지 temp = Block.transaction()를 돌린다.

4. blockchain[id_block].block_data[tx_id]를 새로운 값으로 지정해준다.

```
def change_block(id_block, tx_id):
    tx_id -= 1
    temp = Block.transaction()
    while temp == blockchain[id_block].block_data[tx_id][20:]:
        temp = Block.transaction()
    blockchain[id_block].block_data[tx_id] = str(base58.b58encode(Block.padding(tx_id + 1, 20) +
temp))[2:-1]
```

설명1. change_block(id_block, tx_id)

forgery_or_not(id_block, tx_id)

1. tx_id -= 1

   위와 같은 이유.

2. list_hash는 문제 2번에서 설명한 알고리즘을 이용하여, 블록이 변경되었는지 여부를 확인하는데 필요한 노드들을 모은 리스트이다.

3. new_hash의 초기값은 함수에서 주어진 값에 따라 blockchain[id_block].block_data[tx_id]를 해쉬한 값이다. 그리고 list_hash에 있는 것

모두를 순서대로 해쉬해나간다.

4. original_hash의 초기값은 list_hash[0]이다. 그리고 list_hash에 있는 것 모두를 순서대로 해쉬해나간다.

5. 만약 new_hash와 original_hash의 값이 같다면 not changed를, 그렇지 않다면 changed를 출력한다.

```
def forgery_or_not(id_block, tx_id):
    tx_id -= 1
    list_hash = select_from_tree(id_block, tx_id)

    new_hash = Block.make_hash(blockchain[id_block].block_data[tx_id])
    for i in range(1, len(list_hash)):
        new_hash = Block.make_hash(new_hash + list_hash[i])

    original_hash = list_hash[0]
    for i in range(1, len(list_hash)):
        original_hash = Block.make_hash(original_hash + list_hash[i])

    if new_hash == original_hash:
        print("blockchain[].block_data[]: not changed".format(id_block, tx_id))
    else:
        print("blockchain[].block_data[]: changed".format(id_block, tx_id))
```

설명 2. forgery_or_not(id_block, tx_id)