**Name: Hrishabh Raj** **CS577 MID SEM ASSIGNMENT** **Roll: 1801CS22**

**Q1)**

The Merkle Tree maintains the integrity of the data. If any single detail of transactions or order of the transaction's changes, then these changes are reflected in the hash of that transaction. This change would cascade up the Merkle Tree to the Merkle Root, changing the value of the Merkle root and thus invalidating the block. So everyone can see that Merkle tree allows for a quick and simple test of whether a specific transaction is included in the set or not.

Merkle trees have the following benefits:

1. It provides a means to maintain the integrity and validity of data.
2. It helps in saving the memory or disk space as the proofs, computationally easy and fast.
3. Their proofs and management require tiny amounts of information to be transmitted across networks(Only the hashes are exchanged, not the whole data).
4. Membership of its items can be verified in O(log n) time and space, by just remembering the root hash.

The 'double-spend' problem is prevented in blockchain-based cryptocurrencies such as Bitcoin by using a confirmation mechanism.
Consider the following scenario: Alice has 1 BTC with her. She has malicious intent and is trying to spend that money twice in two separate transactions by sending the same bitcoins to two separate bitcoin wallet addresses.

Now, both of these transactions will then go into the pool of unconfirmed transactions. The first transaction would be approved via the confirmation mechanism and then verified into the subsequent block. However, the second transaction would be recognized as invalid by the confirmation process and would not be verified. If both transactions are pulled from the pool for confirmation simultaneously, the transaction with the highest number of confirmations will be included in the blockchain, while the other one will be discarded.

The above described protocol is used to prevent double spending attacks in bitcoin. But the intended recipient of the second (failed) transaction would not have part in the transaction itself failing, and yet that person would not receive the bitcoin they had anticipated. So, users are advised to for 6 more subsequent blocks being added to the blockchain after their anticipated transaction has been added to a block. Once that is done, users can safely assume that the transaction is valid.

**Q2)**

Given,
To validate a transaction T3 let us check necessary conditions one by one:
Input of T3:

```
"in":[
        {
                "prev_out":{
                        "hash":"h1",
                        "n": 0
```

```
                },
                "scriptSig":"s2 p2"
        },
        {
                "prev_out":{
                        "hash":"h2",
                        "n": 0
                },
                "scriptSig":"s2 p2"
        }
]
```

Output of T3:

```
"out":[
        {
                "value":"15.10",
                "scriptPubKey":"OP_DUP OP_HASH160 <hash of p4>
OP_EQUALVERIFY OP_CHECKSIG"
        }
 ]
```

Transactions T1 and T2 are assumed to be valid(as already present in Blockchain),
T1[0] tells that it transfers 10.12 to p2 publicKey Holder, and T2[0] tells that it transfers 5.00 to p2 publicKey Holder.

Assuming T3[0] to be true, transferring 15.10 to p4 publicKey Holder means,
*# Sum(inputs) > = Sum(outputs) must be True*

1. (hash,n) = (h1,0) = T1[0] = 10.12
2. (hash,n) = (h2,0) = T2[0] = 5.00

Total Input = 10.12 + 5.00 = 15.12
Total output = 15.10
Hence, Total Input >= Total Output.

*# eval(scriptSig + scriptPubKey) must be True*

For the 1st Input of T3,
scriptSig = *“s2 p2”*
scriptPubKey for (hash,n) = (h1,0) = T1[0] = *"OP_DUP OP_HASH160 <hash of p2> OP_EQUALVERIFY OP_CHECKSIG"*
Concatenating them = *“s2 p2 OP_DUP OP_HASH160 <hash of p2> OP_EQUALVERIFY OP_CHECKSIG”*

Evaluation steps of postfix expression implementing through stack:

1. Push ***s2*** into the stack.
2. Push **p2** into the stack.
3. **OP_DUP** will duplicate the top of the stack( p2) & push that in stack.

4. **OP_HASH160** will push the hash of the top of the stack i.e. H(p2).
5. **<hash of p2>** will push H(p2) into the stack again.
6. **OP_EQUALVERIFY** pop two top elements from the stack and compare them.
    a. If they are not equal then the algorithm immediately stops and returns false otherwise continue.
    b. Here, the top two elements are H(p2) and H(p2) which are equal.
7. **OP_CHECKSIG** takes signatures s2 and p2 and verifies them and finally pushes the result into the stack. Here let's consider that the result is true. so true will be pushed in the stack.

The remaining element is the result of the transaction, **here it is true.**
Tabular representation of above mentioned steps:

| Step | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Oprn | s2 | p2 | OP_DUP | OP_HASH160 | <hash of p2> | OP_EQUALVERIFY | OP_CHECKSIG |
| S | | | | | | | |
| T | | | | | H(p2) | | |
| A | | | p2 | H(p2) | H(p2) | | |
| C | | p2 | p2 | p2 | p2 | p2 | |
| K | s2 | s2 | s2 | s2 | s2 | s2 | true |

Similar process will happen for the 2nd Input of T3, returning **True.**
**Thus, the Transaction T3 is valid.**

**Q3)**
In a bitcoin transaction, we have the hash of the block containing the transaction and the index of the transaction in that block. Therefore, the hash of the block can help in retrieving the actual Block.
Since the length of a hashString is fixed and unique for each block, we can store the HashString in a Trie DataStructure, and the end Node will contain a pointer to our Block. Now we can retrieve the value and scriptPubKey to actually validate the Transaction T.

The Time Complexity of finding a dataBlock will take O(m) where *m is the length of the hashString*.
Since there are a large number of Bitcoin transactions, there is also a need to reduce the space complexity of the data structure being constructed for the search operations.
Since these transactions are always between any two accounts only (i.e. One sender and one receiver) and a new Transaction T mostly depends on the input transactions which earlier involved one of the accounts. We can maintain a per-account Trie, which will collect hashes of the dataBlocks, only if a transaction involving that account is present. This will reduce the space required per account basis to manage.

So, first we will create a Trie Data structure per account basis, and traverse the whole Blockchain to populate it.

Whenever any update occurs, we can just update the Trie DataStructure of the accounts involved only, which will also take O(m) Time complexity.