Lab: Merkle Tree

Basic Introduction

Blockchains are prevalent these years, I guess all of you have heard about Bitcoin, which is one of the most famous public blockchain systems.

In Bitcoin system, there are several nodes deployed and every node maintains a ledger which chains a bunch of blocks. Nodes use some consensus protocol to guarantee that all the ledgers record the same blockchains.

Every block has a block head and a block body, block head maintains some metadata, here we only care about **Merkle Root**. And the block body is a transaction list.

The Merkle root is the root value of Merkle Tree. A Merkle tree is a non-linear, binary, hash tree-like data structure. Each leaf node of the tree stores the hash of a transaction in the transaction list, and each non-leaf node stores the hash of the hashes of its two children or stores its child's hash when it only has left child. From the description above, you can figure out the Merkle tree should be constructed from bottom to up. And all the paths from root to leaf have the same length.

Following figure is a Merkle Tree example:

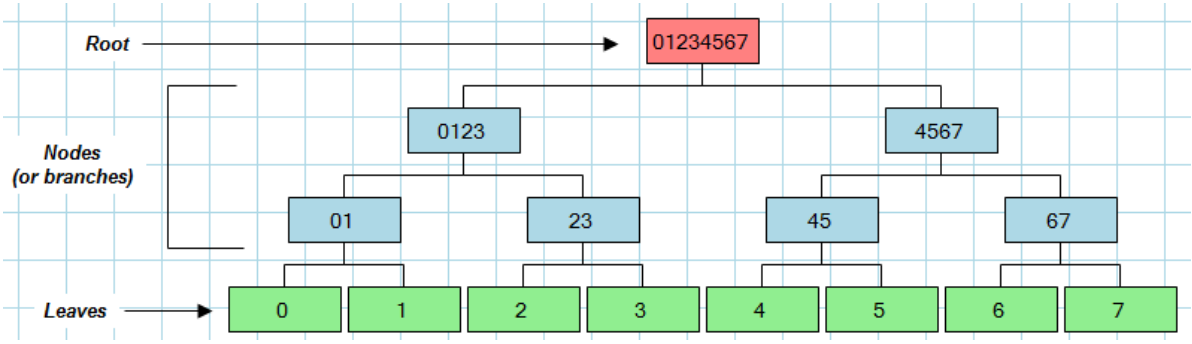


figure 1

Let's think about a question, you just have a transaction record data and a block Merkle root while the ledger is maintained by others. How to verify a transaction, which is to say, how to find out whether a transaction is in the block or not?

In blockchain systems, this verification process can be simplified as the following steps:

- client ask for a transaction proof from the ledger node
- ledger node returns a **ordered** hash value path
- client use the hash value path and the hash of that transaction to re-construct the Merkle Root
- compare the Merkle root re-constructed with the real Merkle Root which is trusted and well-known.
- if the two Merkle root values are the same, client can trust that transaction to be valid

For example, the Merkle Tree is the same as figure 1, and the given hash value is 2, you should return [3, 01, 4567], so the caller can verify whether the hash value 2 is in the tree by constructing the hash value path to get hash root and compare it with the hash root value it already have.

What you need to finish in this lab

Merkle Trees are widely used in block chain systems, they use Merkle Trees to quickly verify a transaction is valid or not. To simplify the problem, you need to construct the Merkle Tree from a transaction hash list, and offer a getProof , addTransaction, getRootHash interfaces.

In your implementation, the Merkle Tree should be a balanced binary tree as you just learned.

Following is the definition of these interfaces:

```
vector<unsigned long> Merkle_Tree::getProof(unsigned long hash_value);
```

 Given a hash value, return a **ordered** hash value tree path and the caller can verify the given value is in the tree, we assume the caller does not has the whole tree but has the root hash value. If the hash value is in the tree, the caller can trust the related transaction.

```
Merkle_Tree::Merkle_Tree();
Merkle_Tree::Merkle_Tree(vector<unsigned long>);
```

These are the two constructors, the first constructs the empty Merkle Tree, the second constructs a Merkle Tree given all the **leaf node** hash values.

```
void Merkle_Tree::addTransaction(unsigned long);
```

Actually, most Merkle Tree does dot need to insert new node, it is just constructed when the block generation.

But for training, we need you to implement an addTransaction.

addTransaction insert a new leaf node to the Merkle Tree, and do not forget to re-compute the hash values of the tree.

To grade your code automatically, in your implementation, you should construct and insert new leaf node in order. More specific, the order of the hash values in the leaf level should be as same as the transaction list and insertion order.

Hint: you can use a linked list to link all the leaf node to support insertion to last level, and maintain a parent pointer to re-compute the hashes.

```
unsigned long Merkle_Tree::getRootHash();
```

getRootHash return the root hash value of the Merkle Tree.

Some restrictions

For simplicity and uniform the result when we test your code

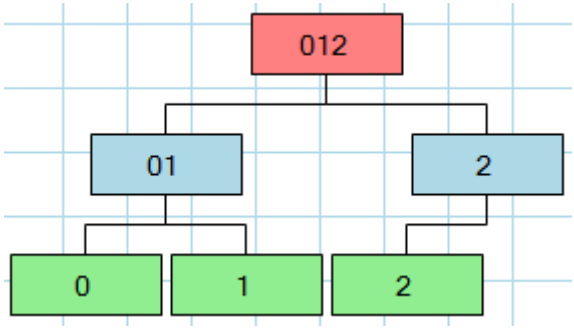
- you should use the hash function we offer to you.
- join the two hash value, (when need to compute parent hash value from its two children) using '+', which is obviously commutative.

Example

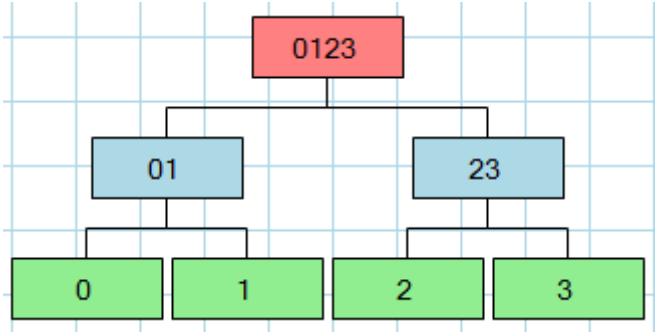
All the numbers in the figures below are hash values rather than id

In the figures, I just merge the two hashes of children to represent, while in your implementation, you should use mathematical '+'.

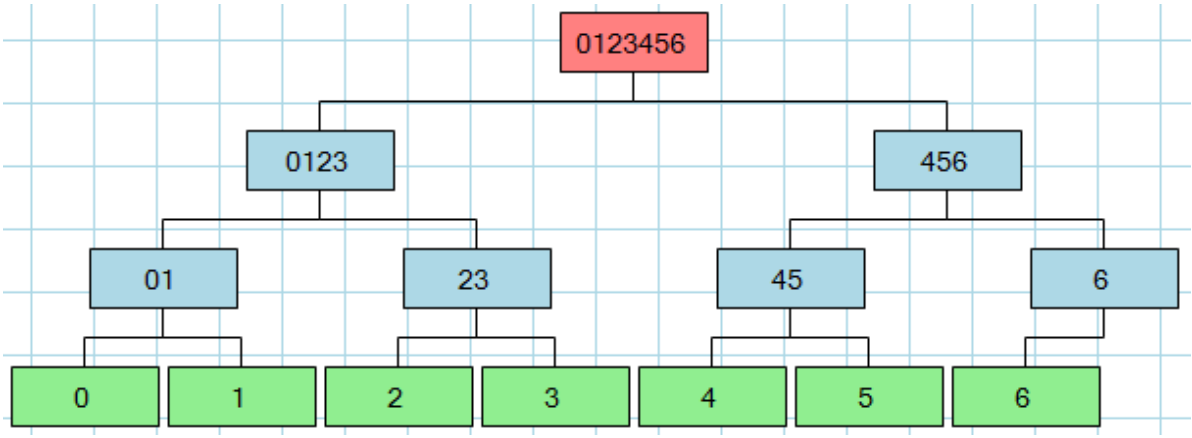
Init Merkle Tree:



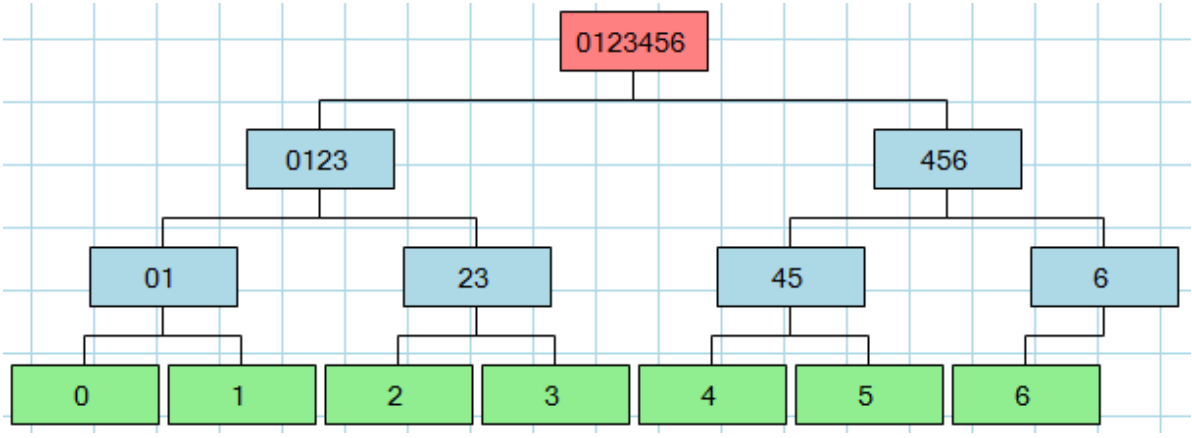
Insert transaction 3:



Insert transaction 4 and 5:



Insert transaction 6:



Submission & Grade

You should implement your Merkle Tree in a single file called `Merkle.cpp`
and upload the CPP file to the canvas platform before the deadline.
Our benchmark tool will check your answer every 30 minutes,
and the result will be posted as comments under your solution.
You can upload as many times as you like before the deadline.