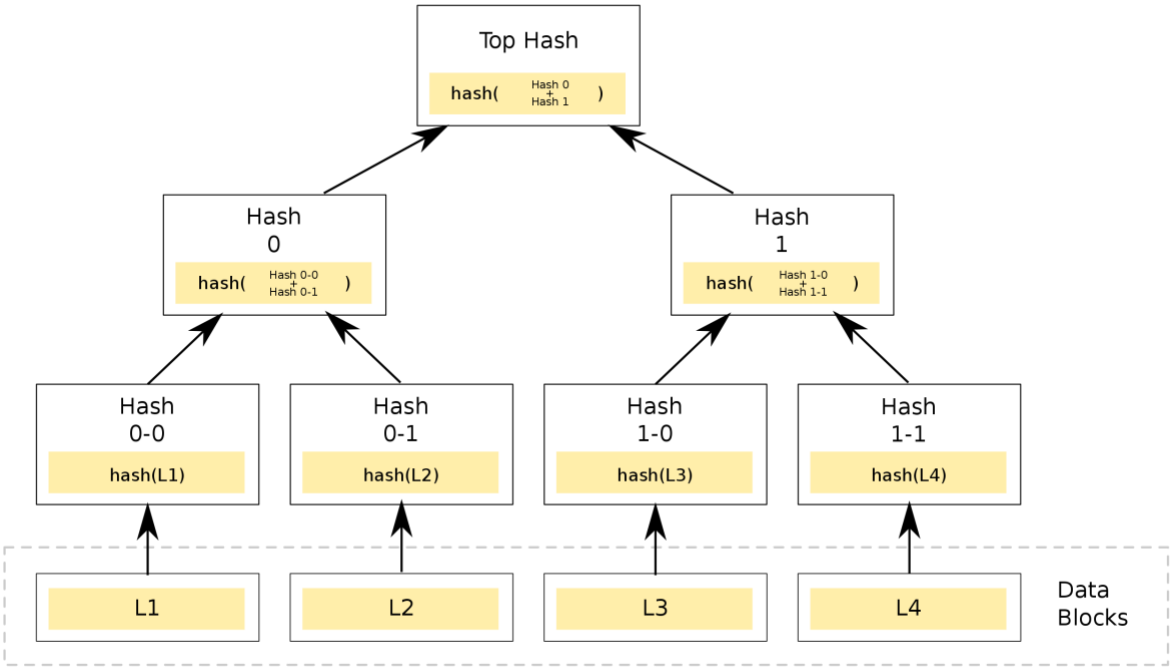
GitHub Link: https://github.com/maverick0701/javaMerkelTree

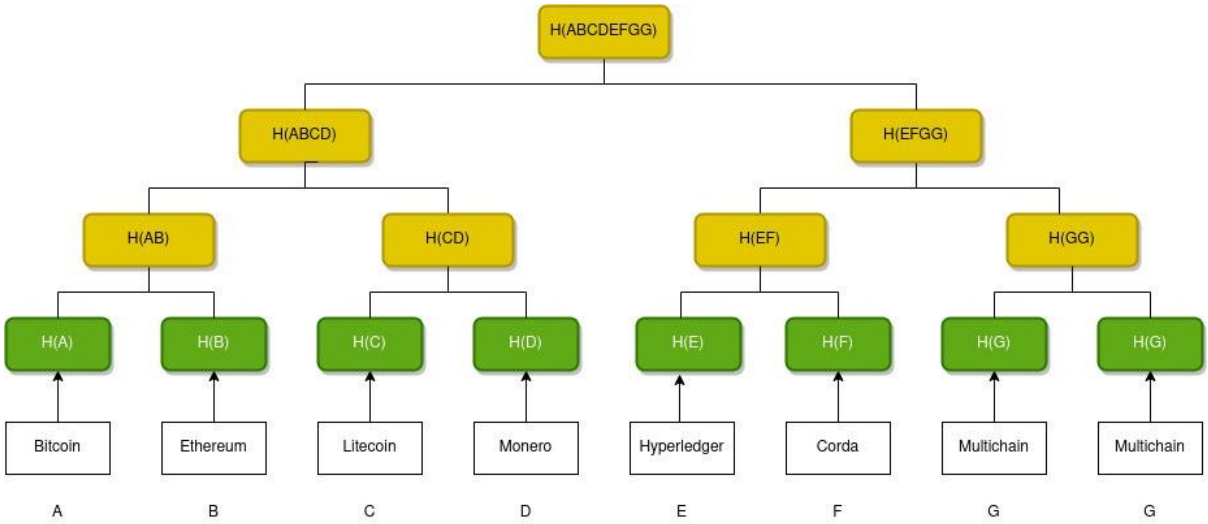
https://github.com/maverick0701/javaMerkelTree

Introduction To Merkel Tree



ARCHITECTURE OF THE

In a Merkle tree, every node is labelled by the hash value of the data it represents. Let us assume that we have a set of data, a list of blockchain networks: *Bitcoin, Ethereum, Litecoin, Monero, Hyperledger, Corda, Multichain*. The Merkle tree will look like below:



Let us now see how the Merkle tree is constructed.

- Find the hash value of every data item. These hash values will be the leaf nodes of our Merkle Tree.
- Group the leaf nodes into pairs, starting from left to right.
- Create a parent node for each pair. Combine the hash values of the leaf nodes and apply the hash function to the combined value. This will be the label of the parent node.
- *Next* group the parent nodes into pairs and repeat the above step.
- Continue the pairing until we are left with a single node which is the Merkle root or root hash.

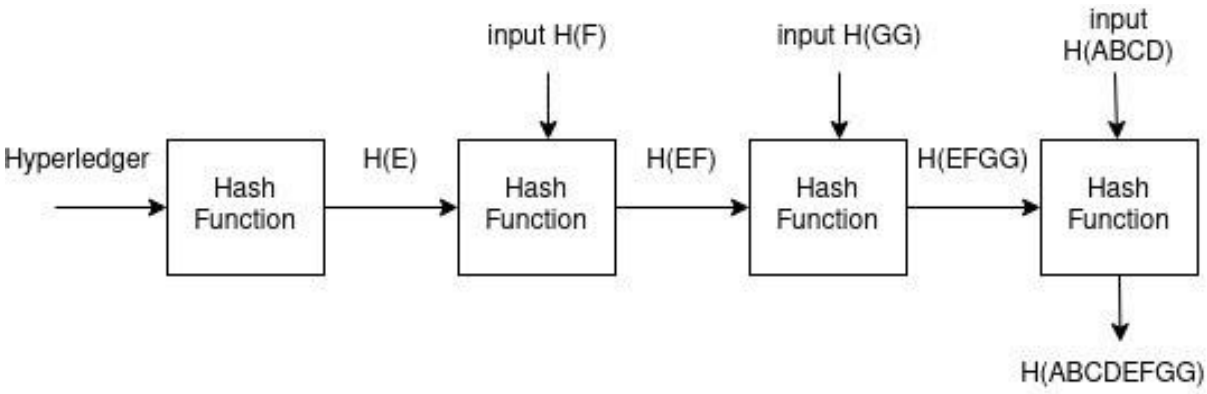
Searching in a Merkle Tree

The major use of the Merkle tree is that it makes search operation faster and efficient. One can easily check whether a

particular data item is represented in the tree, without traversing the entire tree. This is also called **Proof of Inclusion.**

Consider the previous example. Suppose we need to check whether *Hyperledger* is included in the list. The only data we have is the root hash **H(ABCDEFGG)** and **H(E)**, the hash value of "**Hyperledger**". We can re-calculate the value of root hash from **H(E)** by tracing the path from **H(E) to the root node**. *Remember* visiting a node requires a calculation of the hash value represented by that node. For that, we need some additional information.

ere three values are missing to calculate the root hash. If one can get the missing values as input, then it becomes easy to calculate the root hash as shown below.



If you see that the re-calculated value of root is equal to the true root hash, it means that Hyperledger is included in the list. If there is a mismatch, it denotes that Hyperledger is not included in the list.

A *Merkle path* for a leaf in a Merkle tree is the smallest number of additional nodes in the tree required to compute the root hash. In our example, we required 3 additional nodes to calculate the root hash. So the Merkle path for **H(E)** is **H(F)**, **H(GG)**, **H(ABCD)**. We can prove that a data item is included in a Merkle tree by producing a Merkle path. If the root hash calculated using the Merkle path is the same as the true root hash, it simply denotes the leaf node is present in the tree.

How the Merkle Tree Can Guarantee the Security of an Application

the trustworthiness of the records relies on the inability of anyone, including the company that owns the application, to tamper with the records.

An attacker seeks to tamper with the files and to make their record of access to the files undetectable, so their records seem credible. In other data structures where confidentiality is more important, securing the data means hiding it from view.

For data in a Merkle tree to be more secure, it best kept public. Hence, there are several sources of independent consensus on the record of the tree itself. The layers of Merkle tree data structure secure the integrity of the records, and creativity links them with the log of who accessed the records. But, since the Merkle tree makes it near impossible to change anything about any of the elements without the

risk of invalidating all the dependent records up the chain, you can expect the trustworthiness of the records as a set is 99.99% t safe.

Trying to forge a record without a trace would mean finding a collision of a hash of what might be millions of records. Furthermore, a forgery of a full day of records would surface when creating and verifying the mega-meta tree. The security that comes with the use of a Merkle tree allows the users to rest assured that the records they are viewing are accurate.

Why Merkle tree?

The Problem: At the core of the centralized network, data can be accessed from one single copy. This means that they do not have to do much to store or access data. However, when it comes to the decentralized blockchain network, things go haywire as each data is copied among the nodes. So, it is a challenge to efficiently access data. The challenge is also to make a copy of the data and share it among nodes. On top of that, the shared data needs to be verified for each of the receiving nodes.

The Solution: Merkle Trees enable decentralized blockchains to share data, verify them, and make them trustworthy. It organizes data in such a way that not much processing power is needed to share and verify data. It also facilitates the secure transaction thanks to the use of hash functions and cryptography.

Merkle trees are efficient data structures in terms of storage and time. They store hash values of data that require lesser storage space. One can easily search for data by producing the

Merkle path, rather than searching the entire tree. The efficiency of the Merkle tree also depends upon the selection of hash functions.

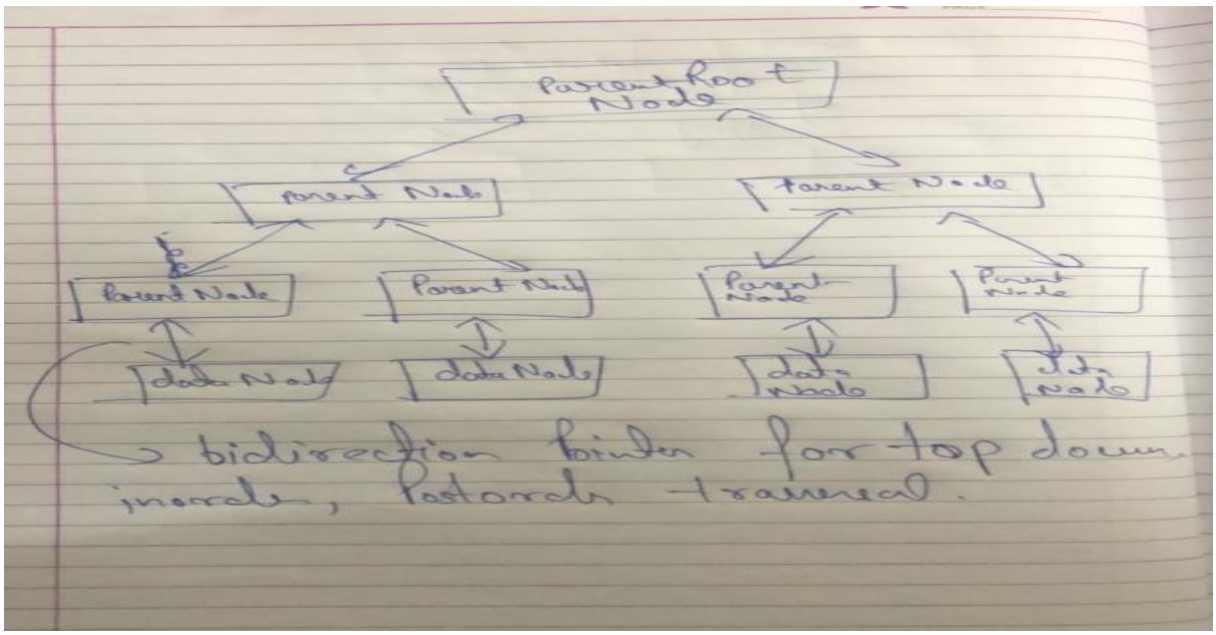
ALGORITHM

Structure of Merkel Node:

```
public class MerkelNode {
    MerkelNode left;
    MerkelNode right;
    String hashText;
    MerkelNode head;
}
```

Head pointer contains refers to the head of current node and its sibling node.
Left and right is analogous to the left and right of binary tree.
Hash text contains sha-512 encryption of the input string.

Class HashAlgo contains function encrypt this string which hashes the input node using sha 512.



STEPS OF ALGORITHM

App.java

1. Start
2. Enter the no blocks you want to store
3. Enter the data string(Example transcation id)
4. Hash the input string store it in a queue for building the tree
5. Make an object of the Merkel Node and call build tree
6. HeadNode of the tree is return
7. Use Headnode to call verify to check whether the node has been tampered with.

Build Tree

Build tree function is based on level wise insertion of nodes from queue until queue is empty.(inspired from Breadth first Search)

1. Store all nodes in a queue.
2. Find the size of the queue and if its odd and not 1 repeat the last node
3. Store current Size of the queue.
4. For(int i=0;i<currentSize;i++)
    a. Pop two nodes from the queue and combine their hash text;
    b. Hash the combined hash text
    c. Make parent node
    d. parentNode.hashText=combinedHash
    e. Insert the parentBNode in the queue;
5. If the queue size is 1
6. Return the node in queue as it is the parent Node.

Verify Function in MerkelNode.

1. Combine the hash of the given node with its sibling
2. Hash the combined hash
3. If it is equal to the parent nodes hash
4. Call recursion on the head node
5. If it is not equal return false

Contains Function Merkel Node

1. Pass the node with root node of the tree

2. Call recursion on left and right child
3. If node == left or right child return true
4. If node.left and right ==null return false

Implementation:

App.java

```
import java.util.ArrayList;
import java.util.Scanner;

import javax.print.DocFlavor.STRING;
public class App {
    public static void main(String[] args) throws Exception {
        ArrayList<MerkelNode>data=new ArrayList<>();
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter The No of Data Block You Want To Store");
        int n=sc.nextInt();MerkelNode lastNode=new MerkelNode();
        for(int i=0;i<n;i++)
        {
            String str=sc.next();
            MerkelNode nd=new MerkelNode(str);
            data.add(nd);
            lastNode=nd;
        }
        if(data.size()%2!=0)
        {

            data.add(lastNode);

        }

        System.out.print("Start Processing"+data.size());
        MerkelNode tree=new MerkelNode();MerkelNode head=tree.BuildTree(data);
        System.out.println("Size of Merel Tree is "+MerkelNode.size(head));
        MerkelNode.inorder(head);

        if(head.doesExsists(data.get(2).hashText, head))
        {
            System.out.println("Yes the node Exists");
```

```
        }
        else
        {
            System.out.println("The node doesnot exist");
        }
    }
}
```

HashAlgo

```
// Java program to calculate SHA-512 hash value
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class HashAlgo {
    public static String encryptThisString(String input)
    {
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-512");
            byte[] messageDigest = md.digest(input.getBytes());
            BigInteger no = new BigInteger(1, messageDigest);
            String hashtext = no.toString(16);
            while (hashtext.length() < 32) {
                hashtext = "0" + hashtext;
            }
            return hashtext;
        }
        catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
}
```

MerkelNode

```
import javax.management.Query;
```

```
import java.util.*;
public class MerkelNode {
   MerkelNode left;
   MerkelNode right;
   String hashText;
   MerkelNode head;
   Queue<MerkelNode>merkelQueue;
   MerkelNode()
   {}
   public static int log2(int size)
   {
      int result = (int)(Math.log(size) / Math.log(2));
      return result;
   }
   MerkelNode(String data)
   {
      this.hashText=HashAlgo.encryptThisString(data);
      this.left=null;
      this.right=null;
      this.head=null;
   }
   public void ProcessQueue(int parentQueueSize)
   {
      for(int j=0;j<parentQueueSize;j++)
         {
            MerkelNode ltNode=this.merkelQueue.poll();
            MerkelNode rtNode=this.merkelQueue.poll();
            String NewData=ltNode.hashText+rtNode.hashText;
            MerkelNode prNode=new MerkelNode(NewData);
            prNode.left=ltNode;
            prNode.right=rtNode;
            ltNode.head=prNode;
            rtNode.head=prNode;
            this.merkelQueue.add(prNode);
         }

   }
   public  MerkelNode BuildTree(ArrayList<MerkelNode> block)
   {
      this.merkelQueue= new LinkedList<>();
```

```
    block.forEach((MerkelNode node)->this.merkelQueue.add(node));
    while(this.merkelQueue.size()!=1)
    {
        int parentQueueSize=this.merkelQueue.size()/2;
        if(parentQueueSize!=1 && parentQueueSize%2==0)
        {
           ProcessQueue(parentQueueSize);
        }
        else if(parentQueueSize!=1 && parentQueueSize%2!=0)
        {
            int newParentQueueSize=parentQueueSize+1;
            // int numElem=(int)Math.pow((double)2,(double)newParentQueueSize);
            MerkelNode repeatNode=this.merkelQueue.peek();
            // while(this.merkelQueue.size()!=numElem)
            // {
            this.merkelQueue.add(repeatNode);
            // }
            ProcessQueue(newParentQueueSize);
        }
        else
        {
            ProcessQueue(parentQueueSize);
            this.merkelQueue.peek().head=this.merkelQueue.peek();
        }
    }
    return this.merkelQueue.peek();
}
public static int size(MerkelNode node)
{
    if(node.left==null && node.right==null)
    {
        return 1;
    }
    return size(node.left) + size(node.right) +1;
}
public boolean verify(MerkelNode node)
{

    String newHash=HashAlgo.encryptThisString(node.left.hashText+node.head.right.hashText);
    if(!newHash.equals(node.hashText))
```

```
            return true;
        if(node.head==node)
        {
            return true;
        }
        return verify(node.head);
    }

    public boolean doesExsists(String node,MerkelNode root)
    {
        if(node==root.hashText)
            return true;
        if(root.left==null && root.right==null)
            return false;
        return doesExsists(node, root.right) || doesExsists(node, root.left);
    }

    public static void inorder(MerkelNode node)
    {
        System.out.println(node.hashText);
        if(node.left==null && node.right==null)
        {
            return;
        }
        inorder(node.left);
        inorder(node.right);
    }

}
```

Output

```
madhavs-MacBook-Pro-2:src madhav$ git add .
madhavs-MacBook-Pro-2:src madhav$ git commit -m 'added doc'
[main 7e142de] added doc
 2 files changed, 6 insertions(+), 3 deletions(-)
 rewrite src/App.class (86%)
madhavs-MacBook-Pro-2:src madhav$ cd "/Users/madhav/Desktop/java/MerkelTree/src/" && javac App.java &
& java App
Enter The No of Data Block You Want To Store
4
transaction1
transactionId2
transactionId3
transactionId4
Start Processing4Size of Merel Tree is 7
b5685b0ce6b5d3cd69c46f937764f699c323b0ff253c2447ec8d550cb5ac66c271dd4ce4c95ee814cd95d5f5f2a9393523815
58f7847d484df5115d14dec11d3
eecc5fd48c1c1c157cd95c50c878490d5e8a460294acb08920b23bce979382506ff7cf620d083c2c58f33df47aaa564d5fcec
98e150d2997ffa2866d32712ae5
d2183975b92856b6e8d4cb0bdaf41ddb97e99e6146a91a5df5997569242075fdb40f6cd82f708ab9b7b05491e4c2a18b18783
fe54d4184175f382b934d6e2387
857b073385445fd9cf3c3442522364192e55b9693d0e97ec7d73cfc312d8874d8ec0a0a2716708705b09d7b9a53f1ec54b27e
a557ab1db6c60a52ff015e61f35
e2bc27e0aa4abeecbc35da17a3ecc53e6975cff3a4824f026571b7b5d84d740c226f5463ad590b3e3ad910a0ff33246c4e410
27ccbfa0921acf0ee0ac79c39fd
23341640600825c5505c254f91fe38ceb4ce22bd8aa9e165a7eb9c269f6fe93bf855bb7bb7b923e6d1eeed5984832141929a7
5043ed8189967d9e49ce62215c5
83791b44b5912ec13aaca273a8590239b946bd351360c02e3b2daac6c790d5f06286b38c81fa11247dba5691010ac620d19c2
03a41a402d150aa70db85bb2b75
Yes the node Exists
madhavs-MacBook-Pro-2:src madhav$
```

Output Explanation:

1.Size of merkel tree is 7

2.All hash of every Node is printed using inorder Traversal.

3.TamperChecking is done

Applications:

- Merkle trees are useful in distributed systems where same data should exist in multiple places.
- Merkle trees can be used to check inconsistencies.
- Apache Cassandra uses Merkle trees to detect inconsistencies between replicas of entire databases.
- It is used in bitcoin and blockchain.

REFRENCES

Bosamia, Mansi, and Dharmendra Patel.
"Current trends and future implementation

possibilities of the Merkel tree." *International Journal of Computer Sciences and Engineering* 6.8 (2018): 294-301.

APA

Aubert, B., Bona, M., Boutigny, D., Karyotakis, Y., Lees, J. P., Poireau, V., ... & Sundermann, J. E. (2008). Observation of Tree-Level B Decays with s s¯ Production from Gluon Radiation. *Physical review letters*, *100*(17), 171803.

Liwei, T., & Yu, S. (2021, May). Research summary of blockchain fragmentation propagation mechanism based on Merkel tree. In *Journal of Physics: Conference Series* (Vol. 1914, No. 1, p. 012010). IOP Publishing.

Merkel, Roger C., Kevin R. Pond, Joseph C. Burns, and Dwight S. Fisher. "Intake, digestibility and nitrogen utilization of three tropical tree legumes: I. As sole feeds compared to Asystasia intrusa and Brachiaria brizantha." *Animal Feed Science and Technology* 82, no. 1-2 (1999): 91-106.