CSE1007

JAVA PROGRAMMING

DA-1

NAME:-SUNNY KUMAR

CLASS:-2nd YEAR B.TECH

REG NO.:-20BCE0262

FACULTY:- PROF. **RA K SARAVANAGURU**

SLOT:- L45+L46

DATE:- 27-04-2022

1. Introduction to Merkle Tree/ Hash Tree

Merkle tree also known as hash tree is a data structure used for data verification and synchronization.
It is a tree data structure where each non-leaf node is a hash of it's child nodes. All the leaf nodes are at the same depth and are as far left as possible.
It maintains data integrity and uses hash functions for this purpose.

Hash Functions:
So before understanding how Merkle trees work, we need to understand how hash functions work.
A hash function maps an input to a fixed output and this output is called hash.
The output is unique for every input and this enables fingerprinting of data.
So, huge amounts of data can be easily identified through their hash.

Applications:

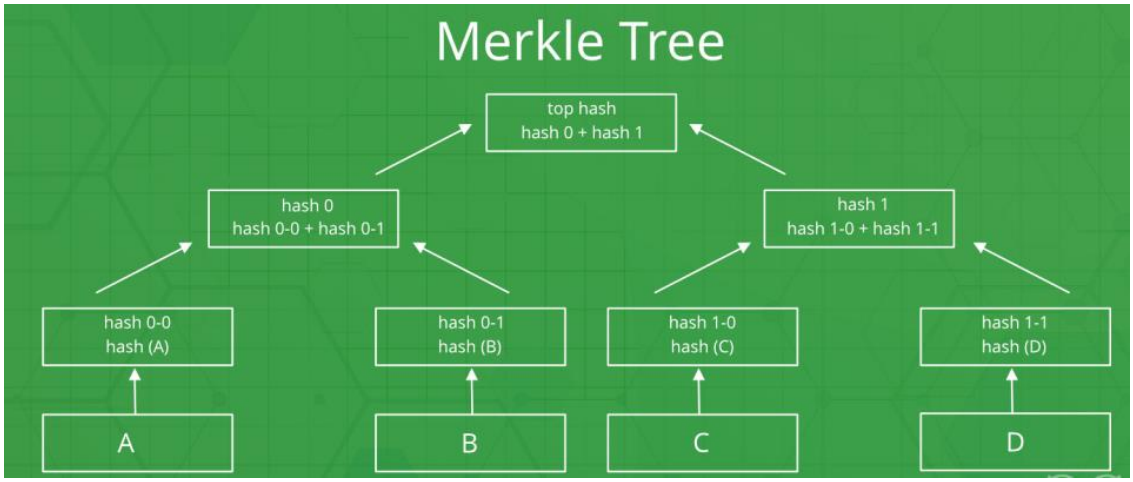
- Merkle trees are useful in distributed systems where same data should exist in multiple places.
- Merkle trees can be used to check inconsistencies.
- Apache Cassandra uses Merkle trees to detect inconsistencies between replicas of entire databases.
- It is used in bitcoin and blockchain.

2. Architecture of Merkle Tree

Structure of the node of binary Merkle tree

It contains four variables:

- It contains a key variable.
- It contains value variable
- It contains two links.



This is a **binary merkel tree**, the top hash is a hash of the entire tree.

- This structure of the tree allows efficient mapping of huge data and small changes made to the data can be easily identified.
- If we want to know where data change has occurred then we can check if data is consistent with root hash and we will not have to traverse the whole structure but only a small part of the structure.
- The root hash is used as the fingerprint for the entire data.

For a Binary Merkel tree

Operation	Complexity
Space	O(n)
Searching	O(logn)
Traversal	O(n)
Insertion	O(logn)
Deletion	O(logn)
Synchronization	O(logn)

3. Algorithm and 4. Illustrate the steps of algorithm

Add operation in Merkle tree

This function is used to **add a node into the merkle tree**. This function is used to insert source code into a git repository. When we commit a change to a repository. Git computes SHA-1 over the contents of that directory tree

and stores them with metadata. The metadata includes information such as pointer to parent commit and a commit message as a commit object.

Algorithm of find function in Merkle tree

Step 1: We will take tree and key as parameters.
Step 2: If the tree is null then we will return null.
Step 3: If the tree->key is equal to the key we will return the tree.
Step 4: If the key is smaller than tree->key then we will return find(tree->left, key)
Step 5: else return find(tree->right, key)

Add operation in Merkle tree

This function is used to **add a node into the merkle tree**. This function is used to insert source code into a git repository. When we commit a change to a repository. Git computes SHA-1 over the contents of that directory tree and stores them with metadata. The metadata includes information such as pointer to parent commit and a commit message as a commit object.
We will create a structure called binary_tree. It contains a pointer to a node called head.

Algorithm to add a node in Merkle tree.

Step 1: We will take key and value as parameters.
Step 2: Take the hash(key) and store it in a variable called index.
Step 3: store (struct node*) arr[index].head in a pointer called tree of datatype node.
Step 4: create a new node with its key as key and value as value and both links as null.
Step 5: If the tree is null then assign the new node to the head and increment the size by 1.
Step 6: If the tree is not null then we will check if the key is already present in the tree using the find function.
Step 7: If the key is already present in the tree then we will update the value.
Step 8: If it is not present in the tree then we will use the insert function to insert the element.

Algorithm of insert function.

Step 1: It will take tree and item pointers of node data type as parameters.
Step 2: If item->key is smaller than tree->key and tree->left is null then assign the item to tree->left.
Step 3: If item->key is smaller than tree->key and tree->left is not null then call insert function with tree->left and item as parameters.
Step 4: If item->key is greater than tree->key and tree->right is null then assign the item to tree->right.
Step 5: If item->key is greater than tree->key and tree->right is not null then call insert function with tree->right and item as parameters.

Delete a node from merkle tree

This function is used to **delete a node from Merkle tree**. If the key given is present in the merkle tree then it will delete the node from the tree. Git remembers all the files you have staged and stores them in a tree structure inside the commit. The nodes of this tree represent your files and directories. This function is used to delete these nodes.

Algorithm to delete a node in Merkle tree.

Step 1: We will take a key as a parameter.
Step 2: Take the hash(key) and store it in a variable called index.
Step 3: store (struct node*) arr[index].head in a pointer called tree of datatype node.
Step 4: If the tree is null then the key is not present.
Step 5: If the tree is not null then we will check if the key is already present in the tree using the find function.
Step 6: If the find function returns null then the key is not present in the tree.
Step 7: If it is not null then we will use the remove function to delete the element.

Algorithm of remove function.

Step 1: It will take tree and key as parameters.
Step 2: If the tree is null then return null.
Step 3: If the key is smaller than the tree->key then tree->left is equal to remove(tree->left, key) and return tree.
Step 4: If the key is greater than the tree->key then tree->right is equal to remove(tree->right, key) and return tree.
Step 5: else if the tree->left is equal to null and the tree->right is equal to null then decrement the size and return tree->left.
Step 6: else if the tree->left is not equal to null and the tree->right is equal

to null then decrement the size and return tree->left.
Step 7: else if tree->left is equal to null and tree->right is not equal to null then decrement the size and return tree->right.
Step 8: else assign tree->left to a pointer called left of data type node.
Step 9: While left->right is not equal to null, left is equal to left->right.
Step 10: tree->key is equal to left->key.
Step 11: tree->value is equal to left->value.
Step 12: tree->left is equal to remove(tree->left, tree->key).
Step 13: Return tree.

5. Create and Verify the merkle tree (Implementation using Java)

Leaf.java

```
package merkletree;

import java.util.List;

/**
 * Represents a Merkle Tree leaf, consisting of a list
 * of blocks of arbitrary data. The arbitrary data in each block
 * is represented as a byte array.
 */
public class Leaf
{
    // The data to be stored in this node
    private final List<byte[]> dataBlock;

    /**
     * Initialises the leaf node, which consists
     * of the specified block of data.
     *
     * @param dataBlock Data block to be placed in the leaf node
     */
    public Leaf(final List<byte[]> dataBlock)
    {
        this.dataBlock = dataBlock;
    }

    /**
     * @return The data block associated with this leaf node
```

```
 */
public List<byte[]> getDataBlock()
{
    return (dataBlock);
}

/**
 * Returns a string representation of the specified
 * byte array, with the values represented in hex. The
 * values are comma separated and enclosed within square
 * brackets.
 *
 * @param array The byte array
 *
 * @return Bracketed string representation of hex values
 */
private String toHexString(final byte[] array)
{
    final StringBuilder str = new StringBuilder();

    str.append("[");

    boolean isFirst = true;
    for(int idx=0; idx<array.length; idx++)
    {
        final byte b = array[idx];

        if (isFirst)
        {
            //str.append(Integer.toHexString(i));
            isFirst = false;
        }
        else
        {
            //str.append("," + Integer.toHexString(i));
            str.append(",");
        }

        final int hiVal = (b & 0xF0) >> 4;
        final int loVal = b & 0x0F;
        str.append((char) ('0' + (hiVal + (hiVal / 10 * 7))));
        str.append((char) ('0' + (loVal + (loVal / 10 * 7))));
    }

    str.append("]");

    return(str.toString());
}
```

```
    /**
     * Returns a string representation of the data block
     */
    public String toString()
    {
        final StringBuilder str = new StringBuilder();

        for(byte[] block: dataBlock)
        {
            str.append(toHexString(block));
        }

        return(str.toString());
    }

}
```

LeafTest.java

```
package merkletree;

import static org.junit.Assert.assertTrue;

import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

/**
 * JUnit tests for the Leaf class.
 *
 */
public class LeafTest
{
    private Leaf leaf;

    private List<byte[]> blocks1and2;

    private String leafString;

    /**
     * @throws java.lang.Exception
```

```
     */
    @Before
    public void setUp() throws Exception
    {
        try
        {
            // Create some data blocks to be assigned to leaf nodes
            final byte[] block1 = {(byte) 0x01, (byte) 0x02, (byte) 0x03,
(byte) 0x04};
            final byte[] block2 = {(byte) 0xae, (byte) 0x45, (byte) 0x98,
(byte) 0xff};

            // Create leaf nodes containing these blocks
            blocks1and2 = new ArrayList<byte[]>();
            blocks1and2.add(block1);
            blocks1and2.add(block2);

            leaf = new Leaf(blocks1and2);

            leafString =  "[01,02,03,04][AE,45,98,FF]";
        }
        catch (Exception e)
        {
            // Should not throw and exception
            assert false;
        }
    }

    /**
     * Test method for {@link merkletree.Leaf#getDataBlock()}.
     */
    @Test
    public final void testGetDataBlock()
    {
        List<byte[]> dataBlock = leaf.getDataBlock();

        assertTrue("Incorrect data block returned",
                   dataBlock.equals(blocks1and2));
    }

    /**
     * Test method for {@link merkletree.Leaf#toString()}.
     */
    @Test
    public final void testToString()
    {
        assertTrue("Leaf equals " + leaf.toString() +
                   " and not " + leafString,
```

```
                leaf.toString().equals(leafString));
    }

}
```

MerkleTree.java

```
package merkletree;

import java.security.MessageDigest;
import java.util.List;

/**
 * Represents a binary Merkle Tree. This consists of two child nodes, and a
 * hash representing those two child nodes. The children can either be leaf
nodes
 * that contain data blocks, or can themselves be Merkle Trees.
 */
public class MerkleTree
{
    // Child trees
    private MerkleTree leftTree = null;
    private MerkleTree rightTree = null;

    // Child leaves
    private Leaf leftLeaf = null;
    private Leaf rightLeaf = null;

    // The hash value of this node
    private byte[] digest;

    // The digest algorithm
    private final MessageDigest md;

    /**
     * Generates a digest for the specified leaf node.
     *
     * @param leaf The leaf node
     *
     * @return The digest generated from the leaf
     */
    private byte[] digest(Leaf leaf)
    {
        final List<byte[]> dataBlock = leaf.getDataBlock();

        // Create a hash of this data block using the
        // specified algorithm
```

```
        final int numBlocks = dataBlock.size();
        for (int index=0; index<numBlocks-1; index++)
        {
            md.update(dataBlock.get(index));
        }
        // Complete the digest with the final block
        digest = md.digest(dataBlock.get(numBlocks-1));

        return (digest);
    }

    /**
     * Initialises an empty Merkle Tree using the specified
     * digest algorithm.
     *
     * @param md The message digest algorithm to be used by the tree
     */
    public MerkleTree(MessageDigest md)
    {
        this.md = md;
    }

    /**
     * Adds two child subtrees to this Merkle Tree.
     *
     * @param leftChild The left child tree
     * @param rightChild The right child tree
     */
    public void add(final MerkleTree leftTree, final MerkleTree rightTree)
    {
        this.leftTree = leftTree;
        this.rightTree = rightTree;

        // Calculate the message digest using the
        // specified digest algorithm and the
        // contents of the two child nodes
        md.update(leftTree.digest());
        digest = md.digest(rightTree.digest());
    }

    /**
     * Adds two child leaves to this Merkle Tree.
     *
     * @param leftChild The left child leaf
     * @param rightChild The right child leaf
     */
    public void add(final Leaf leftLeaf, final Leaf rightLeaf)
    {
```

```
            this.leftLeaf = leftLeaf;
            this.rightLeaf = rightLeaf;

            // Calculate the message digest using the
            // specified digest algorithm and the
            // contents of the two child nodes
            md.update(digest(leftLeaf));
            digest = md.digest(digest(rightLeaf));
        }

        /**
         * @return The left child tree if there is one, else returns
    <code>null</code>
         */
        public MerkleTree leftTree()
        {
            return (leftTree);
        }

        /**
         * @return The right child tree if there is one, else returns
    <code>null</code>
         */
        public MerkleTree rightTree()
        {
            return (rightTree);
        }

        /**
         * @return The left child leaf if there is one, else returns
    <code>null</code>
         */
        public Leaf leftLeaf()
        {
            return (leftLeaf);
        }

        /**
         * @return The right child leaf if there is one, else returns
    <code>null</code>
         */
        public Leaf rightLeaf()
        {
            return (rightLeaf);
        }

        /**
         * @return The digest associate with the root node of this
```

```
 * Merkle Tree
 */
public byte[] digest()
{
    return (digest);
}

/**
 * Returns a string representation of the specified
 * byte array, with the values represented in hex. The
 * values are comma separated and enclosed within square
 * brackets.
 *
 * @param array The byte array
 *
 * @return Bracketed string representation of hex values
 */
private String toHexString(final byte[] array)
{
    final StringBuilder str = new StringBuilder();

    str.append("[");

    boolean isFirst = true;
    for(int idx=0; idx<array.length; idx++)
    {
        final byte b = array[idx];

        if (isFirst)
        {
            //str.append(Integer.toHexString(i));
            isFirst = false;
        }
        else
        {
            //str.append("," + Integer.toHexString(i));
            str.append(",");
        }

        final int hiVal = (b & 0xF0) >> 4;
        final int loVal = b & 0x0F;
        str.append((char) ('0' + (hiVal + (hiVal / 10 * 7))));
        str.append((char) ('0' + (loVal + (loVal / 10 * 7))));
    }

    str.append("]");

    return(str.toString());
```

```
}

/**
 * Private version of prettyPrint in which the number
 * of spaces to indent the tree are specified
 *
 * @param indent The number of spaces to indent
 */
private void prettyPrint(final int indent)
{
    for(int idx=0; idx<indent; idx++)
    {
        System.out.print(" ");
    }

    // Print root digest
    System.out.println("Node digest: " + toHexString(digest()));

    // Print children on subsequent line, further indented
    if (rightLeaf!=null && leftLeaf!=null)
    {
        // Children are leaf nodes
        // Indent children an extra space
        for(int idx=0; idx<indent+1; idx++)
        {
            System.out.print(" ");
        }

        System.out.println("Left leaf: " + rightLeaf.toString() +
                           " Right leaf: " + leftLeaf.toString());

    }
    else if (rightTree!=null && leftTree!=null)
    {
        // Children are Merkle Trees
        // Indent children an extra space
        rightTree.prettyPrint(indent+1);
        leftTree.prettyPrint(indent+1);
    }
    else
    {
        // Tree is empty
        System.out.println("Empty tree");
    }
}

/**
 * Formatted print out of the contents of the tree
```

```
     */
    public void prettyPrint()
    {
        // Pretty print the tree, starting with zero indent
        prettyPrint(0);
    }
}
```

MerkleTreeTest.java

```
package merkletree;

import static org.junit.Assert.*;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

/**
 * JUnit tests for the MerkleTree class.
 */

public class MerkleTreeTest
{
    private MessageDigest md;

    private Leaf leftLeaf, rightLeaf;

    // Merkle tree consisting only of leaves
    private MerkleTree leafTree;

    // Merkle tree consisting of subtrees
    private MerkleTree merkleTree;

    private String leftLeafString, rightLeafString;

    @Before
    public void setUp() throws Exception
    {
        try
        {
```

```
            // Define the message digest algorithm to use
            md = null;
            try
            {
                md = MessageDigest.getInstance("SHA");
            }
            catch (NoSuchAlgorithmException e)
            {
                // Should never happen, we specified SHA, a valid algorithm
                assert false;
            }

            // Create some data blocks to be assigned to left leaf node
            final byte[] block1 = {(byte) 0x01, (byte) 0x02, (byte) 0x03,
(byte) 0x04};
            final byte[] block2 = {(byte) 0xae, (byte) 0x45, (byte) 0x98,
(byte) 0xff};

            // Create left leaf node containing these blocks
            List<byte[]> blocks1and2 = new ArrayList<byte[]>();
            blocks1and2.add(block1);
            blocks1and2.add(block2);

            leftLeaf = new Leaf(blocks1and2);

            leftLeafString = "[01,02,03,04][AE,45,98,FF]";

            // Create some data blocks to be assigned to right leaf node
            final byte[] block3 = {(byte) 0x99, (byte) 0x98, (byte) 0x97,
(byte) 0x96};
            final byte[] block4 = {(byte) 0xff, (byte) 0xfe, (byte) 0xfd,
(byte) 0xfc};

            // Create right leaf node containing these blocks
            List<byte[]> blocks3and4 = new ArrayList<byte[]>();
            blocks3and4.add(block3);
            blocks3and4.add(block4);

            rightLeaf = new Leaf(blocks3and4);

            rightLeafString = "[99,98,97,96][FF,FE,FD,FC]";

            // Create a tree containing only leaves
            leafTree = new MerkleTree(md);
            leafTree.add(leftLeaf, rightLeaf);

            // Create a tree containing two subtrees
            merkleTree = new MerkleTree(md);
```

```
            merkleTree.add(leafTree, leafTree);
        }
        catch (Exception e)
        {
            // Should not throw and exception
            assert false;
        }
    }

    @Test
    public final void testLeftTree()
    {
        MerkleTree leftTree = merkleTree.leftTree();
        Leaf leftLeaf = leftTree.leftLeaf();
        Leaf rightLeaf = leftTree.rightLeaf();

        assertTrue(leftLeaf.toString().equals(leftLeafString));
        assertTrue(rightLeaf.toString().equals(rightLeafString));
    }

    @Test
    public final void testRightTree()
    {
        MerkleTree rightTree = merkleTree.rightTree();
        Leaf leftLeaf = rightTree.leftLeaf();
        Leaf rightLeaf = rightTree.rightLeaf();

        assertTrue(leftLeaf.toString().equals(leftLeafString));
        assertTrue(rightLeaf.toString().equals(rightLeafString));
    }

    @Test
    public final void testLeftLeaf()
    {
        Leaf leftLeaf = leafTree.leftLeaf();

        assertTrue(leftLeaf.toString().equals(leftLeafString));
    }

    @Test
    public final void testRightLeaf()
    {
        Leaf rightLeaf = leafTree.rightLeaf();

        assertTrue(rightLeaf.toString().equals(rightLeafString));
    }

    @Test
```

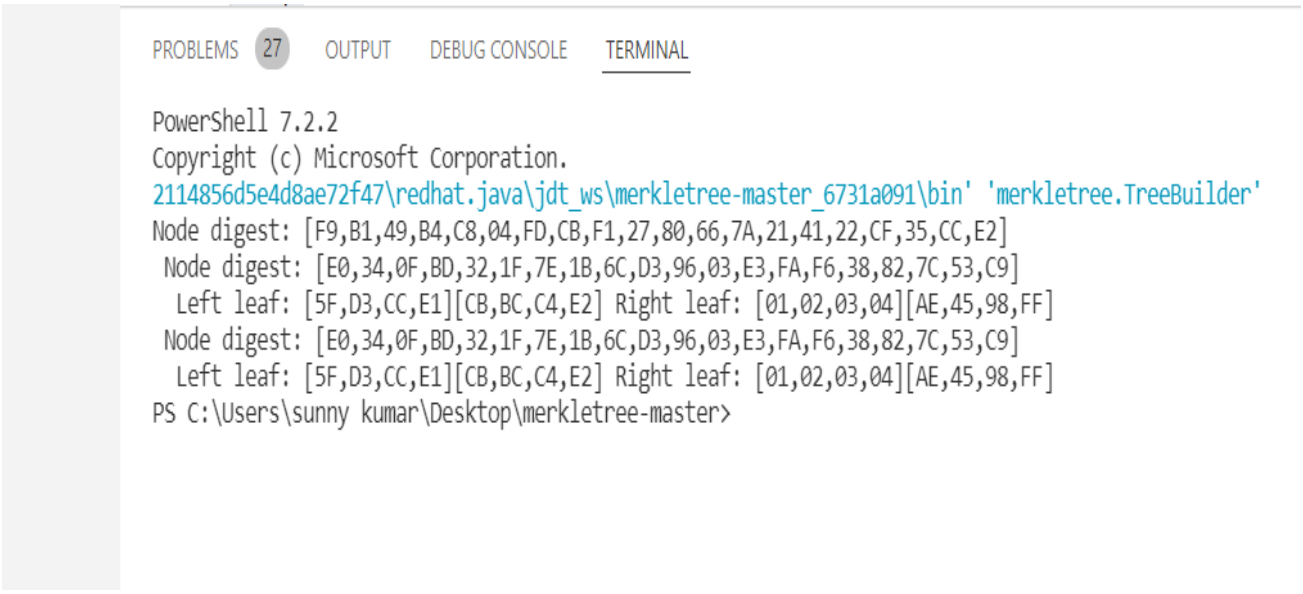
```
    public final void testDigest()
    {
        byte[] merkleTreeDigest = merkleTree.digest();

        byte[] testDigest =
            {(byte) 0xA8, (byte) 0xA3, (byte) 0x36, (byte) 0x1D,
             (byte) 0x40, (byte) 0x1C, (byte) 0xFD, (byte) 0xA3,
             (byte) 0xDD, (byte) 0x26, (byte) 0xF1, (byte) 0x0B,
             (byte) 0xB4, (byte) 0x0E, (byte) 0x1E, (byte) 0xC2,
             (byte) 0xD1, (byte) 0xD8, (byte) 0x94, (byte) 0x6A};

        boolean isValid = true;
        for (int index=0; index<testDigest.length; index++)
        {
            if (merkleTreeDigest[index] != testDigest[index])
            {
                isValid = false;
            }
        }

        assertTrue("Merkle tree digest incorrect", isValid);

        byte[] leafTreeDigest = merkleTree.leftTree().digest();

        byte[] testLeafDigest  =
            {(byte) 0x90, (byte) 0xE0, (byte) 0xE8, (byte) 0xD5,
             (byte) 0xC8, (byte) 0xCC, (byte) 0x1E, (byte) 0x9F,
             (byte) 0xFC, (byte) 0x54, (byte) 0x16, (byte) 0x2A,
             (byte) 0x1D, (byte) 0x2C, (byte) 0xAB, (byte) 0x0A,
             (byte) 0x02, (byte) 0x33, (byte) 0x79, (byte) 0x2B};

        isValid = true;
        for (int index=0; index<testLeafDigest.length; index++)
        {
            if (leafTreeDigest[index] != testLeafDigest[index])
            {
                isValid = false;
            }
        }

        assertTrue("Leaf digest incorrect", isValid);
    }

}
```

TreeBuilder.java

```java
package merkletree;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.List;

import merkletree.Leaf;
import merkletree.MerkleTree;

/**
 * Test class to build and print a Merkle Tree.
 */
public class TreeBuilder
{
    /**
     * Main method creates a simple Merkle Tree consisting of
     * two subtrees, each with two leaf nodes, each of these consisting
     * of two data blocks. Then pretty prints the tree to show its structure.
     *
     * @param noargs
     */
    public static void main(String[] noargs)
    {
        // Define the message digest algorithm to use
        MessageDigest md = null;
        try
        {
            md = MessageDigest.getInstance("SHA");
        }
        catch (NoSuchAlgorithmException e)
        {
            // Should never happen, we specified SHA, a valid algorithm
            assert false;
        }

        // Create some data blocks to be assigned to leaf nodes
        final byte[] block1 = {(byte) 0x01, (byte) 0x02, (byte) 0x03, (byte)
0x04};
        final byte[] block2 = {(byte) 0xae, (byte) 0x45, (byte) 0x98, (byte)
0xff};
        final byte[] block3 = {(byte) 0x5f, (byte) 0xd3, (byte) 0xcc, (byte)
0xe1};
        final byte[] block4 = {(byte) 0xcb, (byte) 0xbc, (byte) 0xc4, (byte)
0xe2};
        final byte[] block5 = {(byte) 0x01, (byte) 0x02, (byte) 0x03, (byte)
0x04};
```

```
        final byte[] block6 = {(byte) 0xae, (byte) 0x45, (byte) 0x98, (byte)
0xff};
        final byte[] block7 = {(byte) 0x5f, (byte) 0xd3, (byte) 0xcc, (byte)
0xe1};
        final byte[] block8 = {(byte) 0xcb, (byte) 0xbc, (byte) 0xc4, (byte)
0xe2};

        // Create leaf nodes containing these blocks
        final List<byte[]> blocks1and2 = new ArrayList<byte[]>();
        blocks1and2.add(block1);
        blocks1and2.add(block2);

        final List<byte[]> blocks3and4 = new ArrayList<byte[]>();
        blocks3and4.add(block3);
        blocks3and4.add(block4);

        final List<byte[]> blocks5and6 = new ArrayList<byte[]>();
        blocks5and6.add(block5);
        blocks5and6.add(block6);

        final List<byte[]> blocks7and8 = new ArrayList<byte[]>();
        blocks7and8.add(block7);
        blocks7and8.add(block8);

        final Leaf leaf1 = new Leaf(blocks1and2);
        final Leaf leaf2 = new Leaf(blocks3and4);
        final Leaf leaf3 = new Leaf(blocks5and6);
        final Leaf leaf4 = new Leaf(blocks7and8);

        // Build up the Merkle Tree from the leaves
        final MerkleTree branch1 = new MerkleTree(md);
        branch1.add(leaf1, leaf2);

        final MerkleTree branch2 = new MerkleTree(md);
        branch2.add(leaf3, leaf4);

        final MerkleTree merkleTree = new MerkleTree(md);
        merkleTree.add(branch1, branch2);

        // Return the digest for the entire tree
        merkleTree.prettyPrint();
    }

}
```

Output:-



6. References

Ralph Merkle, “Secrecy, Authentication and Public Key Systems/ A certified digital signature”, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1979.

Michael Szydlo, “Merkle Tree Traversal in Log Space and Time” (preprint version, 2003).

Markus Jakobsson, Tom Leighton, Silvio Micali and Michael Szydlo, “Fractal Merkle Tree Representation and Traversal”, RSA-CT ’03.

Whitfield Diffie and Martin Hellman, “New Directions in Cryptography”, November 1976.

A. Menezes, P. Van Oorschot and S. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1996.

www.wikipedia.com: “The free online encyclopedia”.

Eli Biham, “Hashing. One–Time Signatures and MACs”, May 2007.