Department of ECE

Vasavi college of Engineering

Hyderabad

Simulation
Implementation of DES

Code :

```
public class DES {

	// initial permutation table

	private static final int[] IP = { 58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36,

				28, 20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32,

				24, 16, 8, 57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19,

				11, 3, 61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7 };


	// inverse initial permutation

	private static final int[] invIP = { 40, 8, 48, 16, 56, 24, 64, 32, 39, 7, 47,

				15, 55, 23, 63, 31, 38, 6, 46, 14, 54, 22, 62, 30, 37, 5, 45, 13,

				53, 21, 61, 29, 36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51,

				19, 59, 27, 34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17,

				57, 25 };


	// Permutation P (in f(Feistel) function)

	private static final int[] P = { 16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5,

				18, 31, 10, 2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4,

				25

	};


	// initial key permutation 64 => 56 bit

	private static final int[] PC1 = { 57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34,

				26, 18, 10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36, 63,

				55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22, 14, 6, 61, 53,

				45, 37, 29, 21, 13, 5, 28, 20, 12, 4 };
```

```
// key permutation at round i 56 => 48

private static final int[] PC2 = { 14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13, 2, 41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29,
            32 };

// key shift for each round
private static final int[] keyShift = { 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2,
            2, 1 };

// expansion permutation from function f
private static final int[] expandTbl = { 32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8,
            9, 10, 11, 12, 13, 12, 13, 14, 15, 16, 17, 16, 17, 18, 19, 20, 21,
            20, 21, 22, 23, 24, 25, 24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32,
            1 };

// substitution boxes
private static final int[][][] sboxes = {
            {           { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7 },
                        { 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8 },
                        { 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0 },
                        { 15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 }
            },
            {           { 15, 1, 8, 14, 6, 11, 3, 2, 9, 7, 2, 13, 12, 0, 5, 10 },
                        { 3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5 },
                        { 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15 },
                        { 13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 }
            },
            {           { 10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8 },
                        { 13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1 },
                        { 13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7 },
                        { 1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 }
            },
        {               { 7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15 },
                        { 13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9 },
                        { 10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4 },
                        { 3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14 }
        },
        {               { 2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9 },
                        { 14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6 },
                        { 4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14 },
                        { 11, 8, 12, 7, 1, 14, 2, 12, 6, 15, 0, 9, 10, 4, 5, 3 }
        },
        {               { 12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11 },
                        { 10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8 },
                        { 9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6 },
                        { 4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13 }


        },
        {               { 4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1 },
                        { 13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6 },
                        { 1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2 },
                        { 6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12 }


        },
        {               { 13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7 },
                        { 1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2 },
                        { 7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8 },
                        { 2, 1, 14, 7, 4, 10, 18, 13, 15, 12, 9, 0, 3, 5, 6, 11 }


        } };

private static byte[][] K;
private static void setBit(byte[] data, int pos, int val) {
    int posByte = pos / 8;

    int posBit = pos % 8;

    byte tmpB = data[posByte];

    tmpB = (byte) (((0xFF7F >> posBit) & tmpB) & 0x00FF);
    byte newByte = (byte) ((val << (8 - (posBit + 1))) | tmpB);
    data[posByte] = newByte;
}

private static int extractBit(byte[] data, int pos) {
    int posByte = pos / 8;
    int posBit = pos % 8;

    byte tmpB = data[posByte];
    return tmpB >> (8 - (posBit + 1)) & 0x0001;
}

private static byte[] rotLeft(byte[] input, int len, int pas) {
    int nrBytes = (len - 1) / 8 + 1;
    byte[] out = new byte[nrBytes];
    for (int i = 0; i < len; i++) {
        int val = extractBit(input, (i + pas) % len);
        setBit(out, i, val);
    }
    return out;
}

private static byte[] extractBits(byte[] input, int pos, int n) {
    int numOfBytes = (n - 1) / 8 + 1;
```

```
        byte[] out = new byte[numOfBytes];
        for (int i = 0; i < n; i++) {
            int val = extractBit(input, pos + i);
            setBit(out, i, val);
        }
        return out;

    }

    private static byte[] permutFunc(byte[] input, int[] table) {
        int nrBytes = (table.length - 1) / 8 + 1;
        byte[] out = new byte[nrBytes];
        for (int i = 0; i < table.length; i++) {
            int val = extractBit(input, table[i] - 1);
            setBit(out, i, val);
        }
        return out;

    }

    private static byte[] xor_func(byte[] a, byte[] b) {
        byte[] out = new byte[a.length];
        for (int i = 0; i < a.length; i++) {
            out[i] = (byte) (a[i] ^ b[i]);
        }
        return out;

    }

    private static byte[] encrypt64Bloc(byte[] bloc,byte[][] subkeys, boolean isDecrypt) {
        byte[] tmp = permutFunc(bloc, IP);
        byte[] L = extractBits(tmp, 0, IP.length/2);
        byte[] R = extractBits(tmp, IP.length/2, IP.length/2);

        for (int i = 0; i < 16; i++) {
            byte[] tmpR = R;
            if(isDecrypt)
                R = f_func(R, subkeys[15-i]);
            else
                R = f_func(R,subkeys[i]);
            R = xor_func(L, R);

            L = tmpR;

        }

        tmp = concatBits(R, IP.length/2, L, IP.length/2);

        tmp = permutFunc(tmp, invIP);
        return tmp;
    }

    private static byte[] f_func(byte[] R, byte[] K) {
        byte[] tmp;
        tmp = permutFunc(R, expandTbl);
        tmp = xor_func(tmp, K);
        tmp = s_func(tmp);
        tmp = permutFunc(tmp, P);
        return tmp;
    }

    private static byte[] s_func(byte[] in) {
        in = separateBytes(in, 6);
        byte[] out = new byte[in.length / 2];
        int halfByte = 0;

        for (int b = 0; b < in.length; b++) {
            byte valByte = in[b];
            int r = 2 * (valByte >> 7 & 0x0001) + (valByte >> 2 & 0x0001);

            int c = valByte >> 3 & 0x000F;
            int val = sboxes[b][r][c];
            if (b % 2 == 0)
                halfByte = val;
            else
                out[b / 2] = (byte) (16 * halfByte + val);
        }
        return out;
    }

    private static byte[] separateBytes(byte[] in, int len) {
        int numOfBytes = (8 * in.length - 1) / len + 1;
        byte[] out = new byte[numOfBytes];
        for (int i = 0; i < numOfBytes; i++) {
            for (int j = 0; j < len; j++) {
                int val = extractBit(in, len * i + j);
                setBit(out, 8 * i + j, val);
            }
        }
        return out;
    }

    private static byte[] concatBits(byte[] a, int aLen, byte[] b, int bLen) {
        int numOfBytes = (aLen + bLen - 1) / 8 + 1;
        byte[] out = new byte[numOfBytes];
        int j = 0;
        for (int i = 0; i < aLen; i++) {
            int val = extractBit(a, i);
            setBit(out, j, val);
            j++;
        }
        for (int i = 0; i < bLen; i++) {
            int val = extractBit(b, i);
            setBit(out, j, val);
            j++;
        }
        return out;
    }

    private static byte[] deletePadding(byte[] input) {
        int count = 0;

        int i = input.length - 1;
        while (input[i] == 0) {
            count++;
            i--;
        }

        byte[] tmp = new byte[input.length - count - 1];
        System.arraycopy(input, 0, tmp, 0, tmp.length);
        return tmp;
    }

    private static byte[][] generateSubKeys(byte[] key) {
        byte[][] tmp = new byte[16][];
        byte[] tmpK = permutFunc(key, PC1);
        byte[] C = extractBits(tmpK, 0, PC1.length/2);
        byte[] D = extractBits(tmpK, PC1.length/2, PC1.length/2);

        for (int i = 0; i < 16; i++) {

            C = rotLeft(C, 28, keyShift[i]);
            D = rotLeft(D, 28, keyShift[i]);

            byte[] cd = concatBits(C, 28, D, 28);

            tmp[i] = permutFunc(cd, PC2);
        }

        return tmp;
    }

    public static byte[] encrypt(byte[] data, byte[] key) {
        int i;
        int length = 8 - data.length % 8;
        byte[] padding = new byte[length];
        padding[0] = (byte) 0x80;

        for (i = 1; i < length; i++)
            padding[i] = 0;

        byte[] tmp = new byte[data.length + length];
        byte[] bloc = new byte[8];

        K = generateSubKeys(key);
```

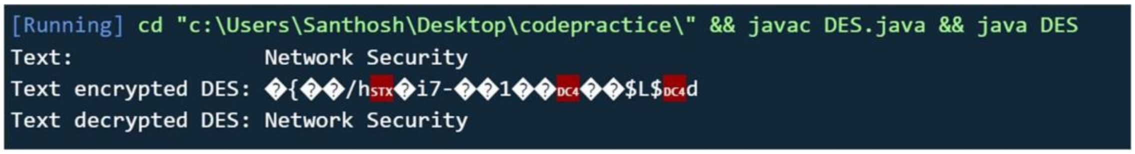
```
        int count = 0;

        for (i = 0; i < data.length + length; i++) {
            if (i > 0 && i % 8 == 0) {
                bloc = encrypt64Bloc(bloc,K, false);
                System.arraycopy(bloc, 0, tmp, i - 8, bloc.length);
            }
            if (i < data.length)
                bloc[i % 8] = data[i];
            else{

                bloc[i % 8] = padding[count % 8];
                count++;
            }
        }
        if (bloc.length == 8) {
            bloc = encrypt64Bloc(bloc,K, false);
            System.arraycopy(bloc, 0, tmp, i - 8, bloc.length);
        }
        return tmp;
    }


    public static byte[] decrypt(byte[] data, byte[] key) {
        int i;
        byte[] tmp = new byte[data.length];
        byte[] bloc = new byte[8];

        K = generateSubKeys(key);

        for (i = 0; i < data.length; i++) {
            if (i > 0 && i % 8 == 0) {
                bloc = encrypt64Bloc(bloc, K, true);
                System.arraycopy(bloc, 0, tmp, i - 8, bloc.length);
            }
            bloc[i % 8] = data[i];
        }
        bloc = encrypt64Bloc(bloc,K, true);
        System.arraycopy(bloc, 0, tmp, i - 8, bloc.length);


        tmp = deletePadding(tmp);


        return tmp;
    }
  public static void main(String[] args) {


        try {
            String text = "Network Security Elective";


            String k = "123456789";
            System.out.println("Text:\t\t\t\t"  +  text);
            byte[] enc = DES.encrypt(text.getBytes(), k.getBytes());
            System.out.println("Text encrypted DES:\t" + new String(enc));


            byte[] dec = DES.decrypt(enc, k.getBytes());
            System.out.println("Text decrypted DES:\t" + new String(dec));


        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Implementation details :

- setBit(byte[] data, int pos, int val): Sets the value of a bit at a specified position in a byte array.
- extractBit(byte[] data, int pos): Extracts the value of a bit at a specified position in a byte array.
- rotLeft(byte[] input, int len, int pas): Performs a left circular rotation on a byte array by a specified number of positions.
- extractBits(byte[] input, int pos, int n): Extracts a specified number of bits from a byte array starting from a given position.
- permutFunc(byte[] input, int[] table): Applies a permutation specified by a table to a byte array.
- xor_func(byte[] a, byte[] b): Performs bitwise XOR operation between two byte arrays.
- encrypt64Bloc(byte[] bloc, byte[][] subkeys, boolean isDecrypt): Encrypts or decrypts a 64-bit block of data using DES algorithm.
- f_func(byte[] R, byte[] K): The Feistel function used in DES.
- s_func(byte[] in): Performs substitution using S-boxes in DES.
- separateBytes(byte[] in, int len): Separates bits into bytes based on a specified length.
- concatBits(byte[] a, int aLen, byte[] b, int bLen): Concatenates two byte arrays into one, considering the specified lengths.
- deletePadding(byte[] input): Deletes padding added during encryption.
- generateSubKeys(byte[] key): Generates 16 subkeys for DES encryption.
- encrypt(byte[] data, byte[] key): Encrypts data using DES.
- decrypt(byte[] data, byte[] key): Decrypts data encrypted using DES.
- main(String[] args): Entry point of the program where encryption and decryption of sample text are demonstrated

In this way the code uses fiestel structure to implement DES based on the standard block Diagram

Results :



```
[Running] cd "c:\Users\Santhosh\Desktop\codepractice\" && javac DES.java && java DES
Text:                 Network Security
Text encrypted DES: �{��/hSTX�i7-��1��DC4��$L$DC4d
Text decrypted DES: Network Security
```

From the above output it is clear that the Encryption and Decryption are working and the Encrypted text is non Ascii so it is showing non ascii symbols.

S. Aruna Deepthi
Assistant Professor,
Department of ECE,
Mail ID: sadeepthi@staff.vce.ac.in

Phone No:9676772232