The Data Structure I used:

The compression algorithm that I wrote uses an array of strings as its main data structure. I used this data structure due to its ease of use, and easy traversal, making the search for duplicates easy. The encode class that I wrote takes an integer matrix of NxM and takes each element and converts to a one-dimensional string array that holds each value. This is done so I can hold special characters that represent a certain degree of repetition.

The string array that I use:

```
//The passed matrix will be saved in this data array and then compressed.
String data[];
```

The actual constructor that assigns the values in the matrix to the string array:

```
//Class constructor
public encode(int[][] matrix)
{
    this.row = matrix.length;
    this.col = matrix[0].length;

    data =  new String[matrix.length * matrix[0].length];
    int place = 0;

    //Take matrix, and convert to a string array.
    for (int row = 0; row < matrix.length; row++)
    {
        for (int col = 0; col < matrix[0].length; col++)
        {
            data[place] = Integer.toString(matrix[row][col]);
            place++;
        }
    }
}
```

The algorithm I used:

My code, once told to compress the string array, checks each value it runs into for duplicates by searching ahead of the current value. If it finds that 3 duplicates exist, it will compress those using a special character. These special characters, represent a certain order of repetition.

Sample compressed 10x10 matrix:

```
Matrix after compression
45 45 22 ! 55 @ 6 @ 28 % 97 $ 3 % 76 * 49 @ 19 ! 71 79 * 76 ! 62 ^ 13 * 36 @ 30 @ 69 % 36 40 40 91 &
```

Each time it runs into a value that can be shrunk, it creates a new array minus the repeated values, resizes the old array, and copies it back.

How I store:
I store each repetition using a special character that represents a certain order of repetition.

The code that does this can be found below:

```
switch(numRepeat)
{
    //If the number of repeats is three.
    case 3:
        //Set the next location to !.
        data[count + 1] = "!";
        break;

    case 4:
        data[count + 1] = "@";
        break;

    case 5:
        data[count + 1] = "$";
        break;

    case 6:
        data[count + 1] = "%";
        break;

    case 7:
        data[count + 1] = "^";
        break;

    case 8:
        data[count + 1] = "&";
        break;

    case 9:
        data[count + 1] = "*";
        break;
}
```

As you can see, each character represents a certain repetition value. In brief, ! = 3, @ = 4, $ = 5, % = 6, ^ = 7, & = 8, * = 9. Note, the current max repetition the algorithm will detect is set to 9. This could be easily changed if more options are added to the switch statement.

How I retrieve:

The original matrix is retrieved by expanding the array of strings. This is done by searching the array of strings for special characters, once one is found, an expansion method is called which adds in a certain number of the same characters. Once fully decompressed, the string is converted back into a matrix, and returned from the method.

Sample call of the expand method:

```
//For the length of the string array.
for (int count = 0; count < data.length; count++)
{
    //Looking for special characters... If found, call the expand function.
    switch (data[count])
    {
        //If !, expand is called with 1, meaning I need to expand the array by 1.
        case "!":
            expand(count, 1);
            //Set count to the next location after the expansion.
            count += 2;
            break;
```

Please see the next page for sample input & output.

Sample input & output:
My code generates random input, meaning that each run is different.

Run 1:

```
Matrix layout before compression:
 8  8  8  8  8 62 62 62 62 62
62 62 45 45 45 45 70 70 70 70
70 70 70 70 52 52 52 52 52 52
52 52 52 49 49 49 49 49 49 49
33 33 33 33 33 33 46 46 46 46
46 46 46 47 47 47 47 47 37 37
37 37 37 37 37 37 37  0  0  0
 0  0  0  0  0  0 46 51 51 51
51 51 51 51 51 51 29 29 29 29
29 29 29 29 29 16 16 16 16 16

Matrix after compression
8 $ 62 ^ 45 @ 70 & 52 * 49 ^ 33 % 46 ^ 47 $ 37 * 0 * 46 51 * 29 * 16 $

Matrix after decompression
 8  8  8  8  8 62 62 62 62 62
62 62 45 45 45 45 70 70 70 70
70 70 70 70 52 52 52 52 52 52
52 52 52 49 49 49 49 49 49 49
33 33 33 33 33 33 46 46 46 46
46 46 46 47 47 47 47 47 37 37
37 37 37 37 37 37 37  0  0  0
 0  0  0  0  0  0 46 51 51 51
51 51 51 51 51 51 29 29 29 29
29 29 29 29 29 16 16 16 16 16

Process finished with exit code 0
```

Run 2:

```
Matrix layout before compression:
76 76 76 76 96 96 96 96 96 96
96 61 61 61 61 61 61 16 16 16
16 16 64 64 64 64 64 64 64 64
87 87 87 87 87  5  5  5  5 82
82 82 82 82 44 44 44 44 44 44
66 66 66 66 18 18 18 18 18 18
18 18 68  0  0  0  0  0 58 58
58 58 58  7  7  7  7  7  7  7
 7 23 23 23 23 23 75 75 75 75
75 95 95 95 95 95 95 95 95 29

Matrix after compression
76 @ 96 ^ 61 % 16 $ 64 & 87 $ 5 @ 82 $ 44 % 66 @ 18 & 68 0 $ 58 $ 7 & 23 $ 75 $ 95 & 29

Matrix after decompression
76 76 76 76 96 96 96 96 96 96
96 61 61 61 61 61 61 16 16 16
16 16 64 64 64 64 64 64 64 64
87 87 87 87 87  5  5  5  5 82
82 82 82 82 44 44 44 44 44 44
66 66 66 66 18 18 18 18 18 18
18 18 68  0  0  0  0  0 58 58
58 58 58  7  7  7  7  7  7  7
 7 23 23 23 23 23 75 75 75 75
75 95 95 95 95 95 95 95 95 29

Process finished with exit code 0
```

Run 3:

```
Matrix layout before compression:
 5  5  5  5  5  5  5  8  8  8
 8  8 27 37 37 37 37 37 23 23
23 23 23 23 23 57 57 75 75 75
75 75 75 75 75 44 44 11 21 21
21 21 21 78 78 78 78 78 78 78
51 51 51 51 51 51 51 51 51 95
95 95 95 95 95 95 95 95 95 95
95 95 95 45 45 45 45 45 45 47
47 47 46 46 46 46 46 46 46 46
38 38 38 38 38 38 38 38 38  5

Matrix after compression
5 ^ 8 $ 27 37 $ 23 ^ 57 57 75 & 44 44 11 21 $ 78 ^ 51 * 95 * 95 $ 45 % 47 ! 46 & 38 * 5

Matrix after decompression
 5  5  5  5  5  5  5  8  8  8
 8  8 27 37 37 37 37 37 23 23
23 23 23 23 23 57 57 75 75 75
75 75 75 75 75 44 44 11 21 21
21 21 21 78 78 78 78 78 78 78
51 51 51 51 51 51 51 51 51 95
95 95 95 95 95 95 95 95 95 95
95 95 95 45 45 45 45 45 45 47
47 47 46 46 46 46 46 46 46 46
38 38 38 38 38 38 38 38 38  5

Process finished with exit code 0
```