

File Compression in C **using Huffman** **Coding**

A lossless data compression
technique

January - 2022

Name : N.Balasubramaniam

Language : C

Aim :

This application aims to compress files to save space in any operating system. The compressed (encoded) files can be later decompressed (decoded) using the same application. As a result lossless decomposition is achieved.

In computer terms the program converts characters to variable length bit strings.

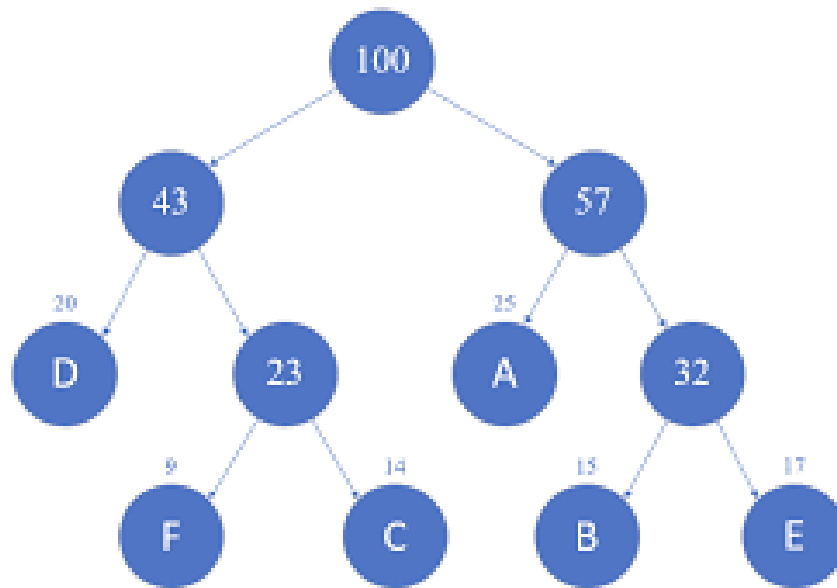
Problem Statement :

To achieve the best case scenario we need the most frequently occurring character to have the least no of bits and least frequently occurring character to have the most no of bits. Each character in the given text file is said to have some frequency (ie. the no of times they appear in the given file).

So we have a set of characters and their frequencies or weights.

The task is to find a prefix free binary code for each of the characters. The code should be in such a way that the character with the highest weight gets the least code length.

This problem is translated to a tree where the leaf nodes are nodes with the characters and we have to find the minimum weighted path from the root of the tree.



Key Functions :

The whole application is splitted into 3 files to implement some sort of modularity and isolation.

main.c

This part of the file includes all the function calls and a menu driven program to call encode and decode functions. It starts with a small intro to Huffman Coding.

encode.h

This header contains all the functions that are used by encode.c . It also contains the definition of structures used as well.

1. Node

This is what makes up the Huffman tree, and contains pointers to left, right, the character and it's frequency.

2.MinHeap

Stores the maximum size of the Heap, and its present size. The array of Nodes is stored as an array in this

3.Codes

Contains character and the huffman code.

4. Freq_map

Contains character and the frequency. Implemented as a Hashmap.

encode.c

This contains all the functions that are used for encoding. It has mainly 4 sub parts :-

- Menu functions
- Helper functions
- Encoding functions
- File Write function

Menu functions give the user with options to use an existing file or new file with new input.

Helper functions perform various operations such as :

1. Waiting for user input
2. Printing binary tree
3. Swapping
4. Printing the queue
5. Swapping nodes
6. Checking leaves

Encode functions can be broadly split into 4 parts :

- Generating the hashmap of frequencies
- Creating the nodes
- Building the huffman tree
- Generating the hashmap of codes

1.encode_begin()

Has all the function calls the all encode functions. Every function is called 1 below the other.

2.encode_frequency()

Creates a hashmap like structure to store the structures and frequencies. All 256 ASCII characters are used. It is later copied to a shorter map where only codes that come in the file are present. Nodes are created using this map using `create_node()` function

3. build_heap()

Creates the heap. It is stored as an array in the `minHeap` structure. Calls the `build_min_heap` function that builds the min heap from the array after heapification.

4. build_min_heap()

Builds the min heap. Moves the child up until the root node is reached and the heap property is satisfied by calling the `heapify` function.

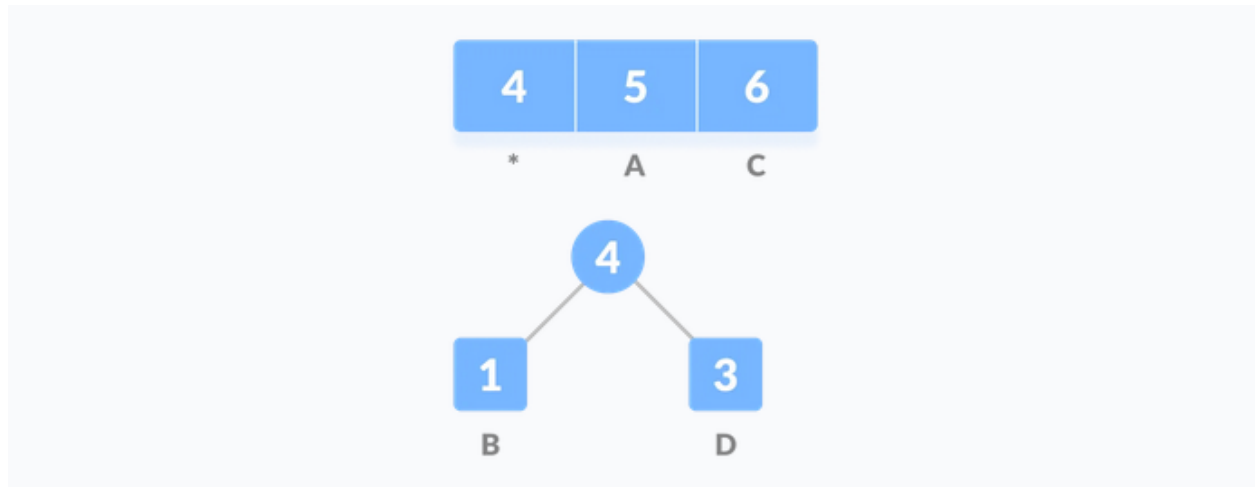
5. heapify()

Start from the first index of a non-leaf node whose index is given by $n/2 - 1$. Set current element i as least. The index of the left child is given by $2i + 1$ and the right child is given by $2i + 2$. If the left child is lesser than the current element (i.e. element at i th index), set `leftChildIndex` as least. If `rightChild` is less than element at least, set `rightChildIndex` as the least. Swap least with the current element.

Thus a priority queue like structure is established.

6. enocde_huffman_tree()

Pops the last 2 priority queue and creates a new node with frequency as sum of their frequencies. This new node is inserted into the priority queue. The resultant priority queue is printed. This process is done till 1 node is left.



7.pop()

Pops the 1st element in the priority queue.

8.insert_heap()

Inserts nodes to the queue based on the priority.

9.print_huffman_tree()

The resulting huffman tree is printed by recursively traversing the tree.

10.reach_leaf_nodes()

An array is maintained which stores whether a traversal is done towards the left or towards the right. If it is done towards the left 0 is pushed to the array. If it is done towards the right 1 is pushed to the array. Once a leaf node is reached

it calls the `encode_map` function which stores the array as a string to the new hashmap.

11.encode_map()

This takes the array, and traverses it. If 1 is present it copies 1 to the empty string otherwise 0. After which the character and the code is copied to the hashmap aside it's corresponding ASCII value.

12.encoding_done()

Marks completion of the encoding process and prints bits saved, bits used etc.

Thus the encoding is complete and now has to be stored in a file.

Encoding in files

A total of 4 files are created.

1. compressed.dat : has the compressed data in binary form. This is the actual compressed file
2. codes.dat: has the character, and it's code, with which decompression takes place
3. compressed.txt: a visual representation of how the binary file is actually stored, showing the bits
4. codes.txt : a visual representation of how the hashmap is stored

1.encoding_to_files()

All 4 files are created here. The input text file is traversed, and the corresponding codes are copied. If C is to be encoded as say 101, then in the hashmap C would be present at the 67th position which could be easily derived by type casting char to int. Hence a $O(1)$ solution is derived.

Similarly the codemap is also copied to the files(the char and the code) and total no of characters(for binary files).

A total count of binary, alphabet, characters are counted separately.

2.WriteBit()

This is the function that actually helps us to write bit by bit to file. Otherwise every 1 or 0 that would be added would amount to 1 byte instead of 1 bit thus increasing the size 8 times, and defeats the purpose of this algorithm.

Bits are collected and bit operations are used to set the individual bits in a character. Once 8 bits are reached it prints the ascii value to the file.

3.Flush_Bits()

There are times where 8 bits are not fully present and hence 0s are added at the end till 8 bits are present and the resulting byte is printed in the file.

Thus we have successfully created 2 files where compressed.dat has the data of input file, and uses less

space and codes.dat has the info required to decompress this file

decode.h

This header contains all the functions that are used by decode.c . It also contains the definition of structures used as well.

1.codes1

This structure is used to store the info from the codemap which is codes.dat. It stores the character along with the corresponding code.

decode.c

This part of the program uses the info from codes.dat to uncompress the file compressed.dat. It has 3 main functions, which is to

- Generate the short map
- Decode codes.dat
- Decode compressed.dat

1. decode_file()

Gets the file names as input. It reads codes.dat and generates the map. It also reads the total no of bits to read which is written in codes.dat. This then calls the generate_short_map() function that stores these details in a hashmap. It then prints those codes to show them to the user. After decode_file_2() is called, it reads the string. It checks if the string matches with any of the codes. If it doesn't it adds the next char to it. If it does it prints the character to uncompressed.txt.

2. generate_short_map()

This function generates the short_map from a map of 256 characters for faster traversal when decoding.

3.decode_file_2()

Reads compressed.dat and writes 1s and 0s to a string. A byte is read and char_to_binary() function is called which converts the given ASCII code to 1s and 0s.

4.char_to_binary()

Reads a char, and then adds 1 or 0 to the string depending on the bit. The 1st bit is accessed through the <<i operator. 'i' is then decremented thus accessing the bits from left to right and copying to string.

Makefile

As this application uses many c files and headers, it is all linked and compiled with help of a Makefile.

One can run it on Linux or a Windows system with Make with the command :

```
make;  
./app
```

If both the options are not viable it can be compiled and linked easily by copying the below text to a terminal

```
gcc -c main.c  
gcc -c encode.c  
gcc -c decode.c  
gcc -o app main.o encode.o decode.o
```

Data Structure used and its functions :

1. Binary Tree

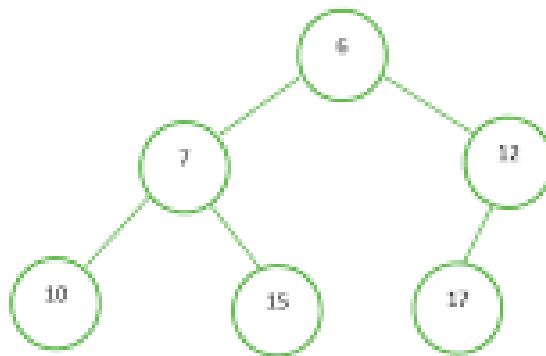
- A binary tree is a tree data structure in which each node has at most two children.

- Here it is used to create the Heap like Data Structure to store the frequencies and the character
- It is also used to implement the Huffman tree.

2. Heap

- A Heap is a special Tree-based data structure in which the tree is a complete binary tree
 - Here a minheap is used, so that we get the least frequencies first to add the node. Hence a maxheap is not viable in this case.
 - This helps it function like a priority queue with appropriate functions.
 - All the nodes are stored in the heap and heapified till 1 single node remains
-

Min-Heap



3. Priority Queue

- An application of minheap where the code with max priority is inserted in the start(least frequency) and vice versa
- This helps us to get elements from the tree in $O(1)$ time, since it is always the root node.
- Hence time complexity is reduced, and enables faster compression.

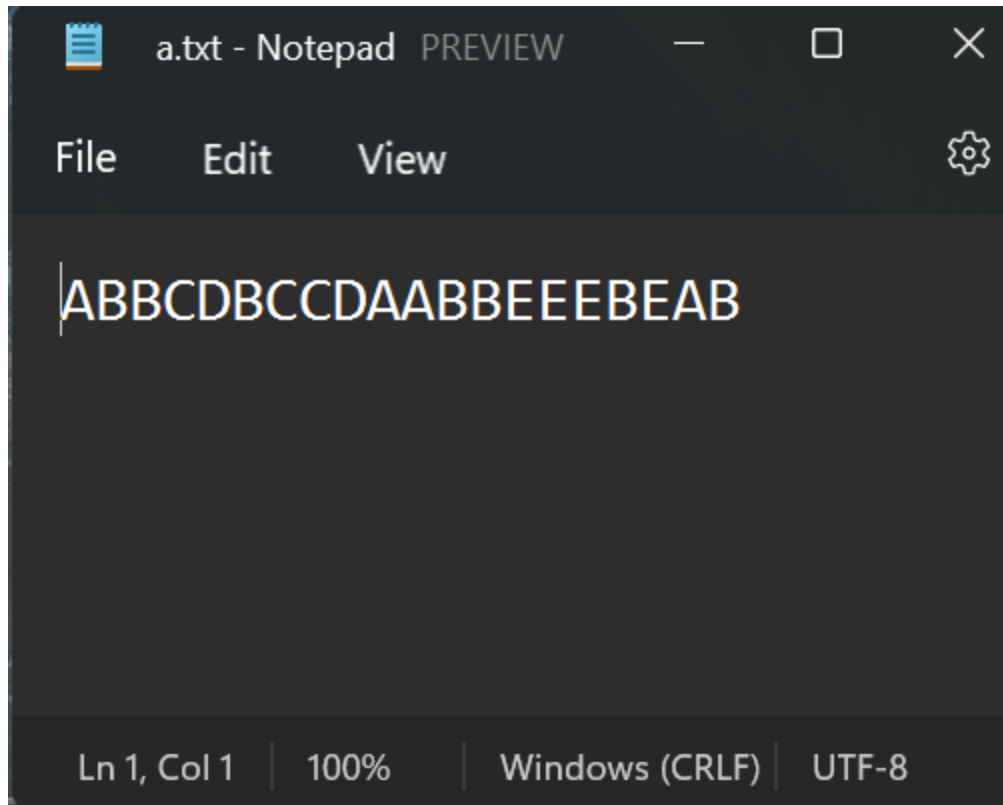
4. Hash Map

- This is one of the most important data structure, as it helps us create the maps, and building the frequencies.
- We can get the codes and frequencies of any character in $O(1)$ time, since we can directly access it by it's ASCII value. This is done with the help of type casting.

Output Screenshots :

Encoding

a.txt :



The image shows a Notepad application window with a dark theme. The title bar at the top reads "a.txt - Notepad PREVIEW" and includes standard window controls (minimize, maximize, close). Below the title bar is a menu bar with "File", "Edit", and "View" options, and a settings gear icon on the right. The main text area contains the string "ABBCDBCCDAABBEEEBEAB" in white monospace font, with a vertical cursor at the beginning. At the bottom, a status bar displays "Ln 1, Col 1", "100%", "Windows (CRLF)", and "UTF-8".

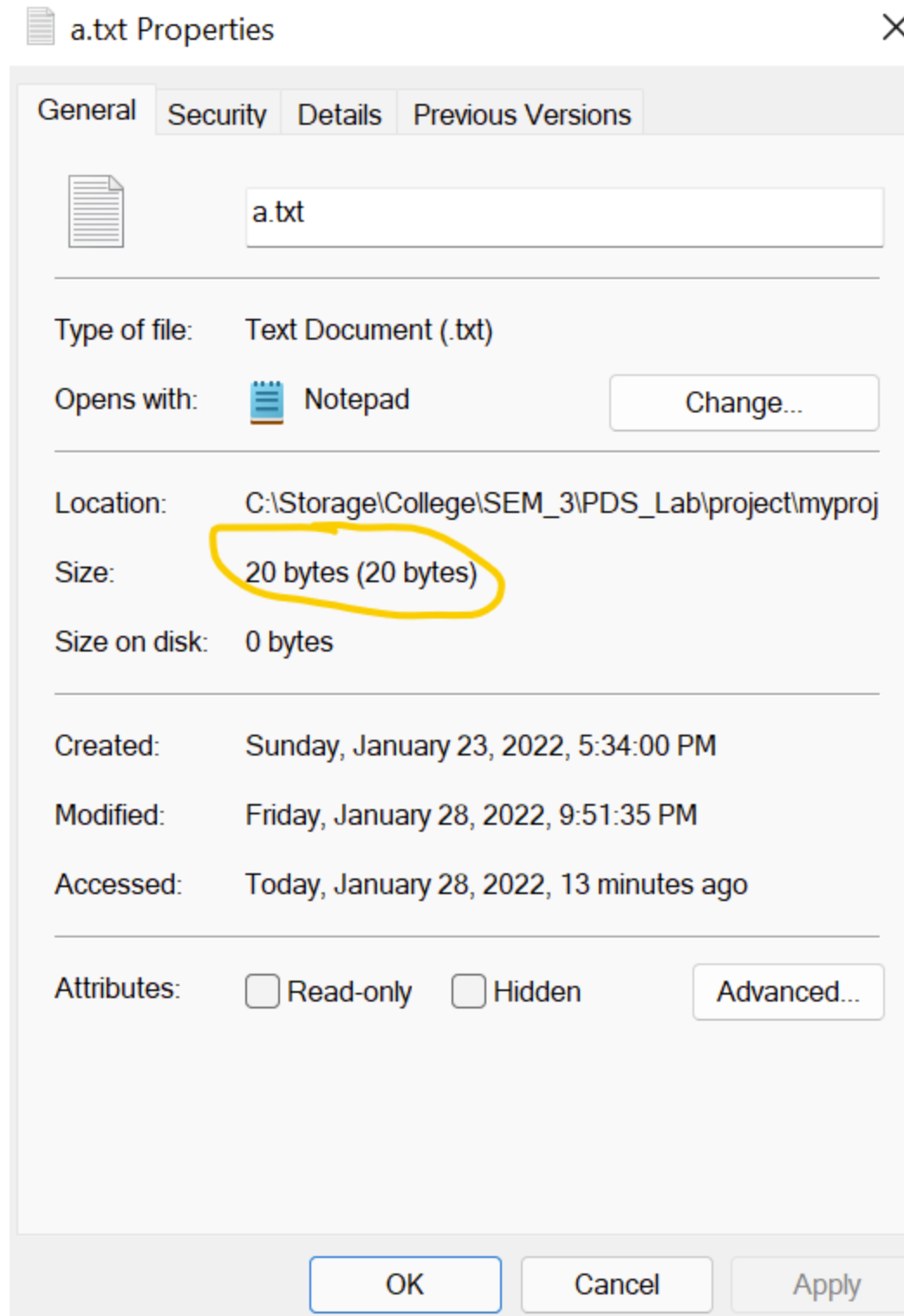
```
a.txt - Notepad PREVIEW
```

File Edit View

ABBCDBCCDAABBEEEBEAB

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Input file details :



Main menu :


```

=====>                Huffman Coding                <=====

- Lossless Data Compression technique
- Encoding follows the prefix rule
- Reduces cost of transmission
- Variable sized encoding used
- Time complexity  $O(n \cdot \log(n))$ 

=====>    Main Menu    <=====

1.Encode a file
2.Decode a file
3.Exit
Enter choice : 1

```

Encode Menu :

```

=====>    Encode Menu    <=====

1.Enter text for a new file
2.Use existing file
3.Go back to main menu
Enter your choice : 2

=====>    Encode File    <=====

Enter the file name : a.txt

File successfully opened
Reading file contents

ABBCDBCCDAABBEEEEBEAB

File read complete

Press enter to continue.....|

```

Hashmap:

```
Total no of characters : 20
File size : 20 bytes

Do you want to continue with the compression (Y/N) :Y

====> Encoding File <====

File name : a.txt

The hashmap of characters and their frequencies are :

A - 4
B - 7
C - 3
D - 2
E - 4
```

After heapification :

```
Press enter to continue.....

After Heapification :

D - 2
A - 4
C - 3
B - 7
E - 4

Press enter to continue.....
```

Building the huffman tree by popping and inserting :

Building the huffman tree

Current Queue : 2 4 3 7 4

Press enter to continue.....

Popped nodes frequency : 2, 3

New node frequency : 5

Current Queue : 4 5 4 7

Press enter to continue.....

Popped nodes frequency : 4, 4

New node frequency : 8

Current Queue : 5 7 8

Press enter to continue.....

Popped nodes frequency : 5, 7

New node frequency : 12

Current Queue : 8 12

Press enter to continue.....

New node frequency : 20

Current Queue : 20

Press enter to continue.....

Thus only 1 node is present and the tree is built.

The huffman Tree :

Huffman Tree :

```

20
├──8
│   ├──4
│   ├──4
│   └──12
│       ├──5
│       ├──2
│       ├──3
│       └──7

```

The corresponding Huffman Code :

Corresponding Huffman code for character :

A	-	00
B	-	11
C	-	101
D	-	100
E	-	01

```
Press enter to continue.....
```

File encoding begins :

```
Encoding to files.....  
Encoded file name : compressed.dat  
Code map for decoding present in : codes.dat  
Press enter to continue.....  
The visual representation of the binary files can be found in the below text files  
Encoded text file name : compressed.txt  
Code map for viewing present in : codes.txt  
Press enter to continue.....|
```

Encoded file is written to file and printed to console, and compression details are mentioned :

```
Encoded file contents :  
001111101100111011011000000111101010111010011  
  
===>Encoding Summary<===  
Total no of characters in the file : 20  
Total no of bits in the uncompressed file : 160  
Total no of bits in the compressed file : 45  
Total no of bits in code map : 21  
Total no of bits after compression :  $45 + 21 = 66$   
Compression ratio : 41.25%  
Encoding completed successfully....
```

Encoding is done :

```

=====>      Encode Menu      <=====





1.Enter text for a new file
2.Use existing file
3.Go back to main menu
Enter your choice : 3
Going back to main menu

=====>      Main Menu      <=====

1.Encode a file
2.Decode a file
3.Exit
Enter choice : 3
Program terminating successfully

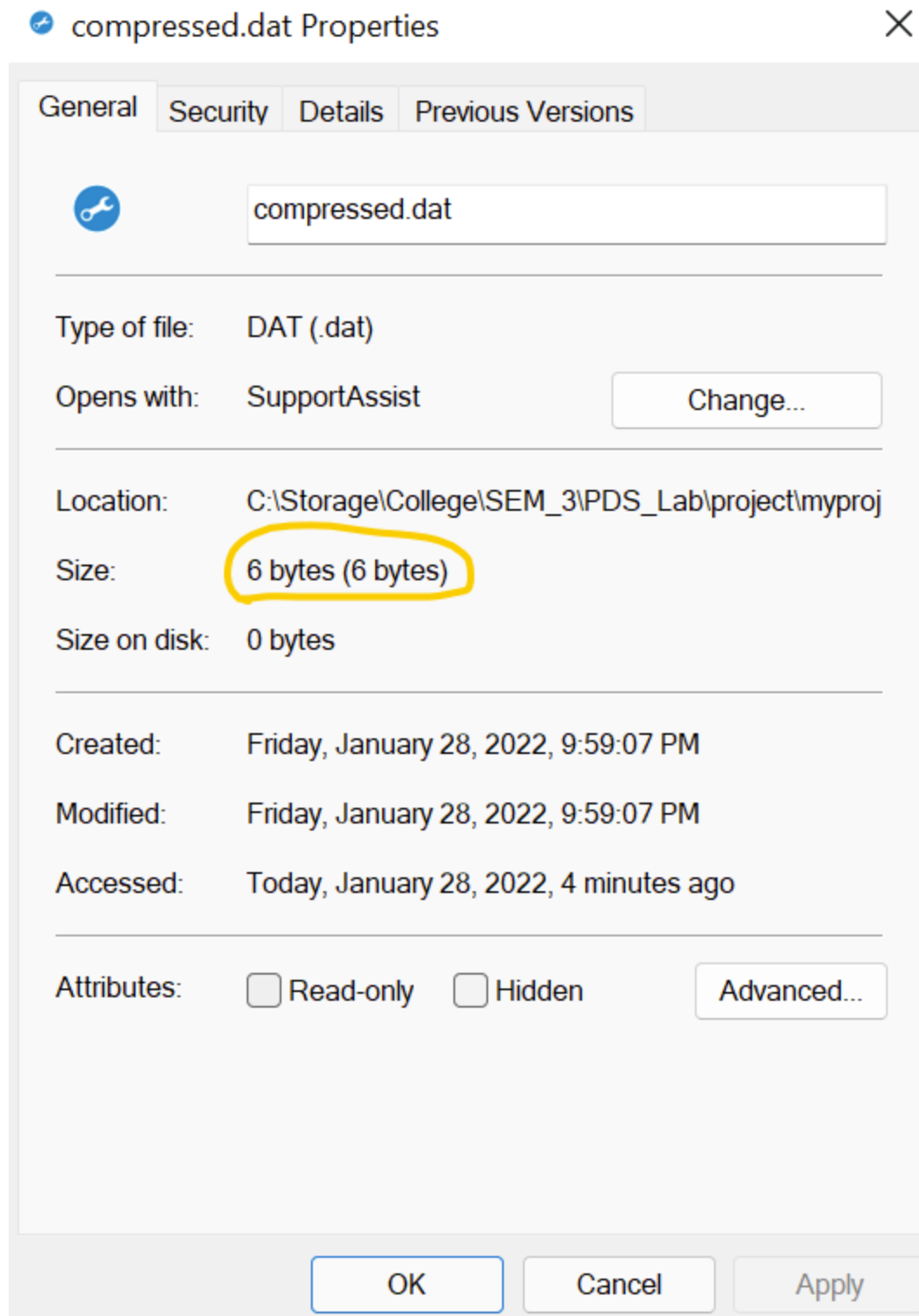
```

Files created :

 codes.dat	1/28/2022 9:59 PM	DAT	1 KB
 codes.txt	1/28/2022 9:59 PM	Text Document	1 KB
 compressed.dat	1/28/2022 9:59 PM	DAT	1 KB
 compressed.txt	1/28/2022 9:59 PM	Text Document	1 KB

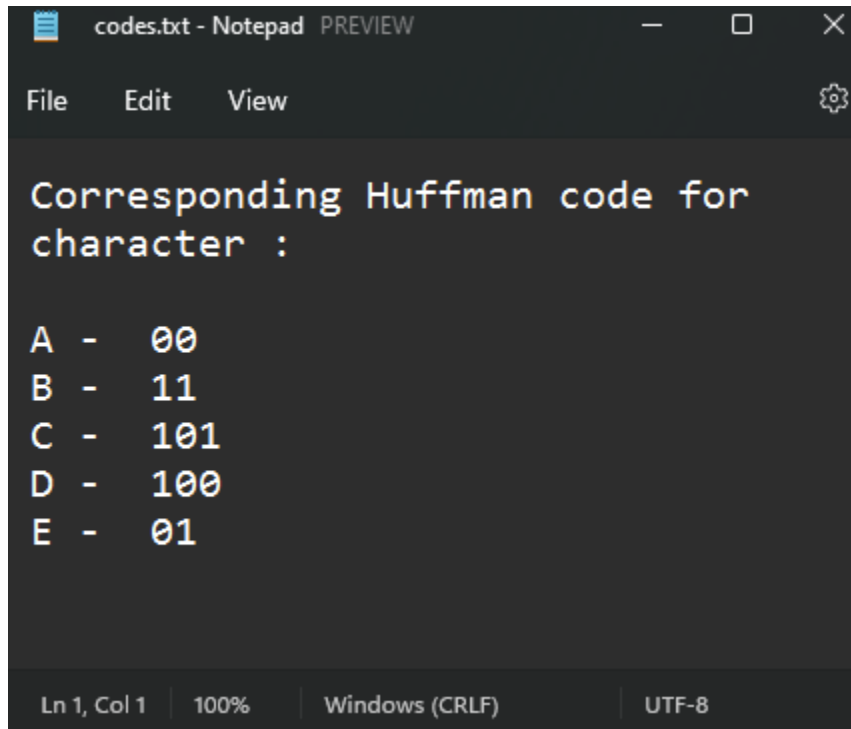
It shows 1 kb for all because that's the minimum that file explorer could show.

But looking at our compressed file through properties :



So initially the file was 20 bytes but now it's 6 bytes after compressing.

codes.txt:

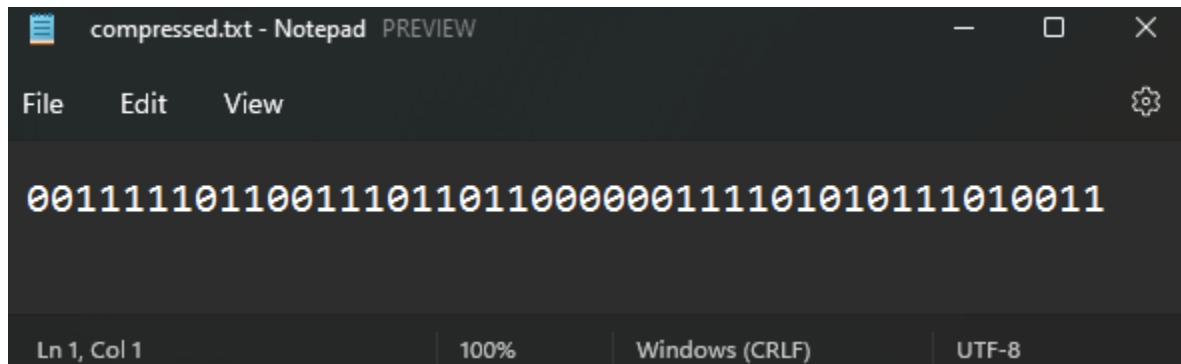


```
codes.txt - Notepad PREVIEW
File Edit View
Corresponding Huffman code for
character :

A - 00
B - 11
C - 101
D - 100
E - 01

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8
```

compressed.txt:



```
compressed.txt - Notepad PREVIEW
File Edit View
001111101100111011011000000111101010111010011

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8
```

Decoding:

Decode menu:


```
=====>      Decode Menu      <=====

1.Enter filename to decode

2.Go back to main menu

Enter your choice : 1

=====>  Decode File      <=====

Enter file name of code map (dat): codes.dat
```

Codes read from binary file and successfully copied to hashmap

```
Total no of bits to be read : 45

Total no of unique characters = 5

Corresponding Huffman code for characters are :

A - 00
B - 11
C - 101
D - 100
E - 01
```

Now the other compressed file is read, and is decoded and its contents are copied to uncompressed.txt



```
Enter file to decode (dat): compressed.dat

Decoded file name : uncompressed.txt

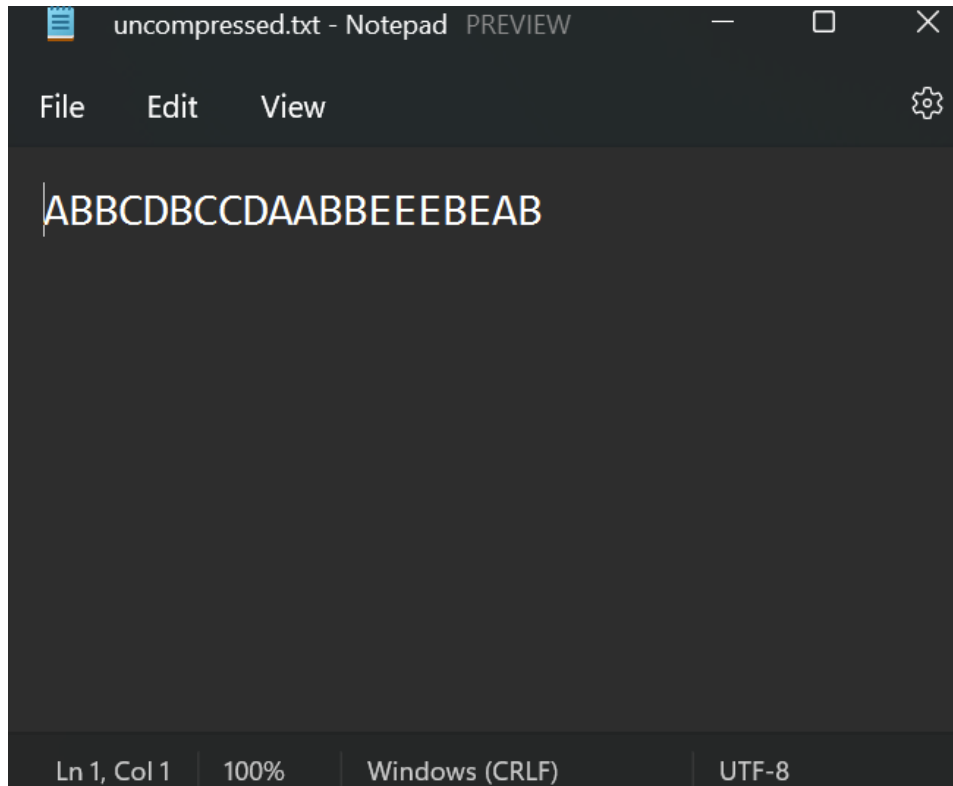
Decoded file contents :
001111101100111011011000000111101010111010011

Uncompressed file :
ABBCDBCCDAABBEEEBEAB
```

The new file is created :

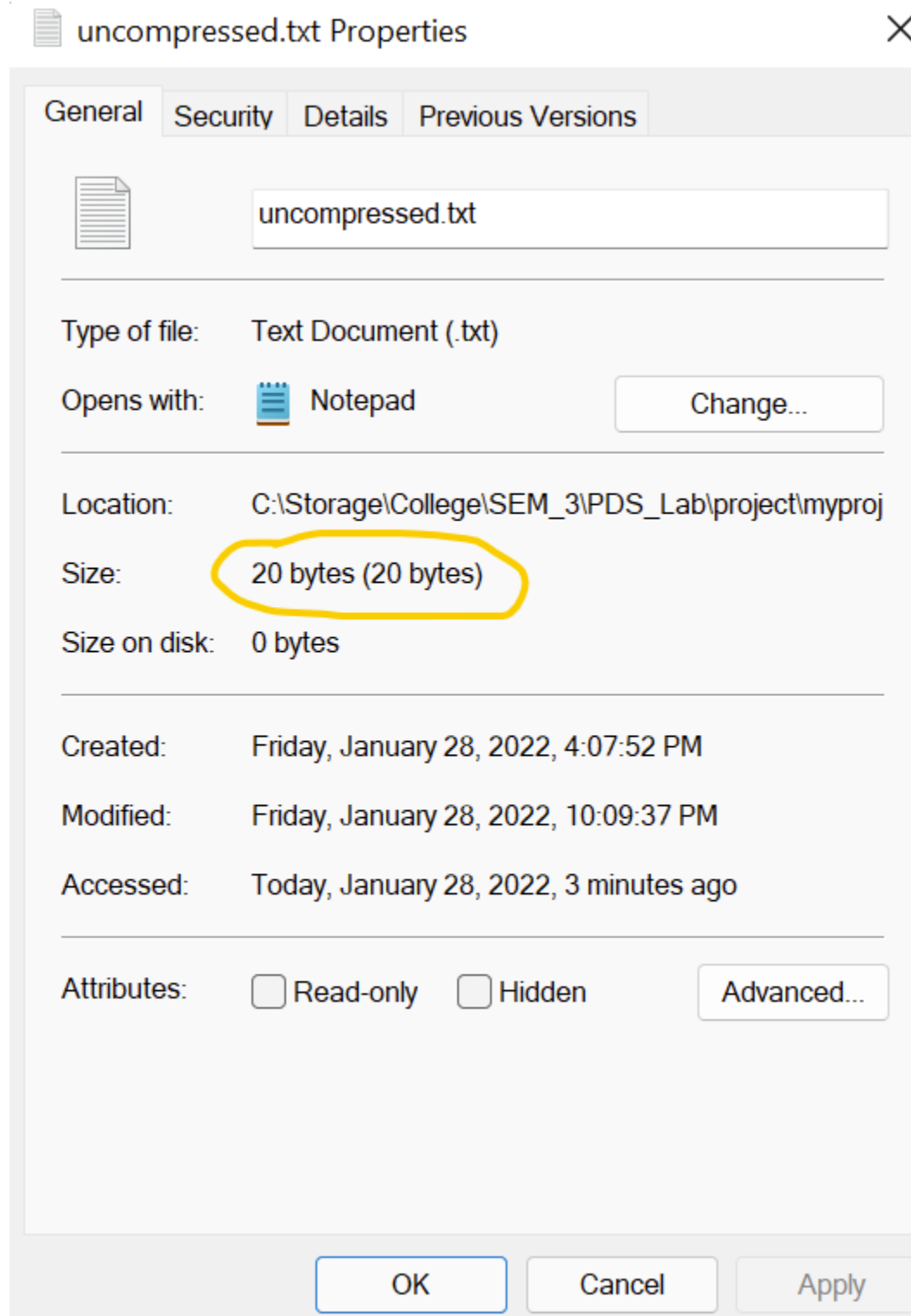
 Makefile	1/23/2022 1:16 PM	File	1 KB
 uncompressed.txt	1/28/2022 10:09 PM	Text Document	1 KB

The new file:

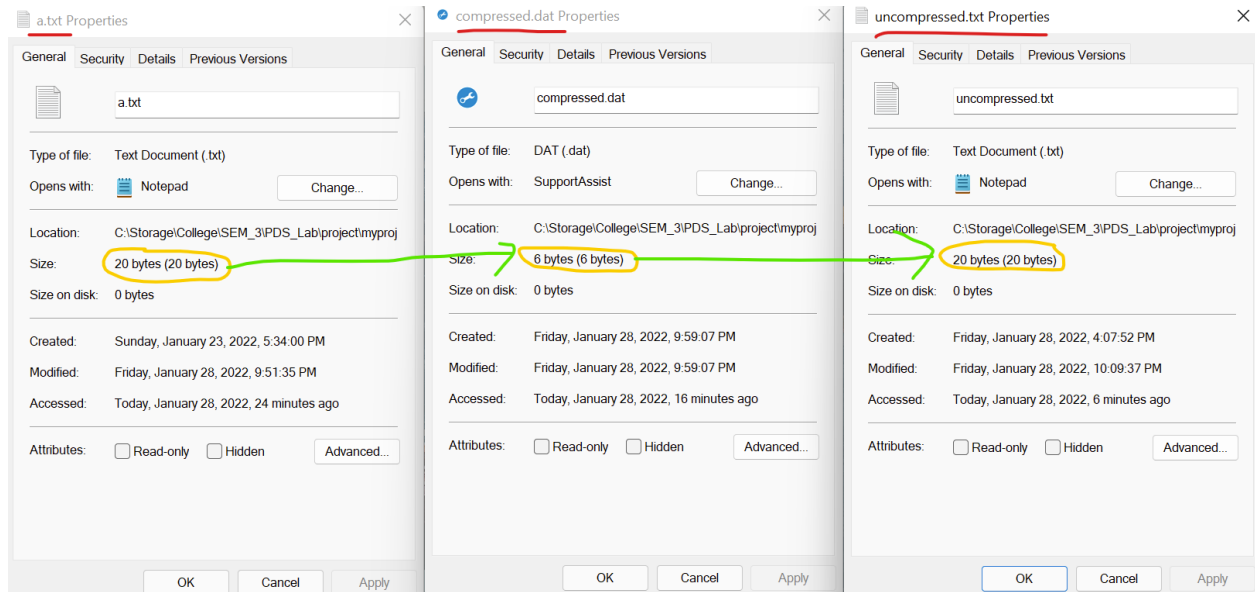


```
uncompressed.txt - Notepad PREVIEW
File Edit View
ABBCDBCCDAABBEEEBEAB
Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8
```

As one can see the uncompressed file is successfully decoded back and the original 20 bytes is restored.



Overall view of the process



Example 2 : Entering new file with console..

```
=====> New File <=====

Enter new file name : b.txt

File created

Enter string to add to file :

Hi my name is Balasubramaniam

Do you want to continue with the compression (Y/N) :Y
```

Huffman Tree :

```
29
├---13
│   ├──6
│   │   ├──3
│   │   │   ├──2
│   │   │   │   ├──1
│   │   │   │   └---1
│   │   └---4
│   │       ├──2
│   │       │   ├──1
│   │       │   └---1
│   │       └---2
│   │           ├──1
│   │           └---1
└---16
    ├──8
    │   ├──4
    │   │   ├──2
    │   │   │   ├──1
    │   │   │   └---1
    │   └---4
    │       ├──2
    │       │   ├──1
    │       │   └---1
    └---8
        ├──4
        │   ├──2
        │   │   ├──1
        │   │   └---1
        └---4
```

Corresponding Huffman code for character :

```
- 101
B - 11110
H - 01101
a - 00
b - 01111
e - 10010
i - 010
l - 11111
m - 110
n - 1000
r - 01100
s - 1110
u - 01110
y - 10011
```

Encoded file name : compressed.dat

Code map for decoding present in : codes.dat

Press enter to continue.....

The visual representation of the binary files can be found in the below text files

Encoded text file name : compressed.txt

Code map for viewing present in : codes.txt

Press enter to continue.....

Encoded file contents :

011010101011101001110110000011010010101011101011111000111110011100111001111011000011000100001000110

```
===>Encoding Summary<===
```

```
Total no of characters in the file : 29
```

```
Total no of bits in the uncompressed file : 232
```

```
Total no of bits in the compressed file : 101
```

```
Total no of bits in code map : 77
```

```
Total no of bits after compression :  $101 + 77 = 178$ 
```

```
Compression ratio : 76.72%
```

```
Encoding completed successfully....
```

Decoding

```
Enter file name of code map (dat): codes.dat
```

```
Total no of bits to be read : 101
```

```
Total no of unique characters = 14
```

Corresponding Huffman code for characters are :

```
- 101
B - 11110
H - 01101
a - 00
b - 01111
e - 10010
i - 010
l - 11111
m - 110
n - 1000
r - 01100
s - 1110
u - 01110
y - 10011
```

Enter file to decode (dat): compressed.dat

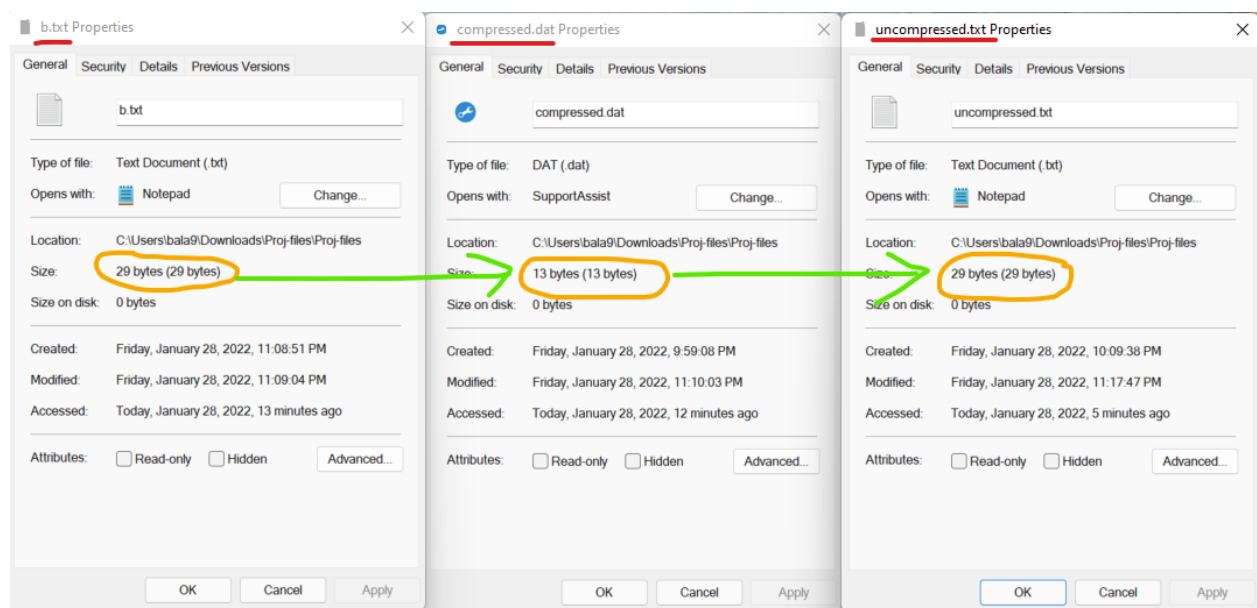
Decoded file name : uncompressed.txt

Decoded file contents :

```
011010101011101001110110000011010010101011101011111000111110011100111001111011000011000100001000110
```

Uncompressed file :

Hi my name is Balasubramaniam



Result:

Hence successful compression and decompression has taken place. This process can be further expanded to

compressing images, folders etc. It has thus been shown that Huffman encoding and decoding is a lossless and successful data compression and decompression algorithm.