Huffman encoder/decoder

Huffman Encoding:

Encoding Scheme: The scheme is based on variable length HUFFMAN coding. It makes use of Greedy algorithm. Root of the tree has intermediate node. Traversing through the tree using code generated leads to Leaf, (Characters in the original file). Byte information is read from the file, based on its ASCII value, the binary string symbol is fetched. The bitstream strings created is then stored as bit information on a BYTE. Details are below

Key points:

1) Reading input: Used Hash table to store characters in the input file. The Data structure used had a array of pointers of basic node of HashMap. So effectively, only characters which were present in the file were instantiated using Heap. No entries with 0 frequencies were ever created.

```
____Huffman Hashtable____
S.no   Key    Value   Count
______|________|______|____
10    10
     1
32    32         4999
44    44    ,    666
46    46    .    773
59    59    ;    12
65    65    A     49
67    67    C     76
68    68    D     62
69    69    E     33
70    70    F     45
73    73    I    46
76    76    L    4
77    77    M     49
78    78    N     95
80    80    P     151
81    81    Q     22
83    83    S     73
85    85    U     21
86    86    V     72
97    97    a     2144
98    98    b     375
99    99    c     1104
100   100   d      714
```

```
101    101    e      3266
102    102    f     186
103    103    g      398
104    104    h      170
105    105    i     2696
106    106    j     30
108    108    l     1722
109    109    m      1234
110    110    n      1649
111    111    o      1154
112    112    p      666
113    113    q      334
114    114    r     1515
115    115    s     2352
116    116    t     2204
117    117    u      2622
118    118    v     384
121    121    y     19
```

2) Created an array of Huffman node to be sorted to get frequencies.
3) Sorted the array of Huffman node using Radix sort. Radix sort makes use of Count Sort.

```
Max Elemental Item Count = 4999
Sorted entries 0= 1 Ascii = 10
Sorted entries 1= 4 Ascii = 76
Sorted entries 2= 12 Ascii = 59
Sorted entries 3= 19 Ascii = 121
Sorted entries 4= 21 Ascii = 85
Sorted entries 5= 22 Ascii = 81
Sorted entries 6= 30 Ascii = 106
Sorted entries 7= 33 Ascii = 69
Sorted entries 8= 45 Ascii = 70
Sorted entries 9= 46 Ascii = 73
Sorted entries 10= 49 Ascii = 65
Sorted entries 11= 49 Ascii = 77
Sorted entries 12= 62 Ascii = 68
Sorted entries 13= 72 Ascii = 86
Sorted entries 14= 73 Ascii = 83
Sorted entries 15= 76 Ascii = 67
Sorted entries 16= 95 Ascii = 78
Sorted entries 17= 151 Ascii = 80
Sorted entries 18= 170 Ascii = 104
Sorted entries 19= 186 Ascii = 102
Sorted entries 20= 334 Ascii = 113
Sorted entries 21= 375 Ascii = 98
Sorted entries 22= 384 Ascii = 118
Sorted entries 23= 398 Ascii = 103
Sorted entries 24= 666 Ascii = 44
Sorted entries 25= 666 Ascii = 112
Sorted entries 26= 714 Ascii = 100
Sorted entries 27= 773 Ascii = 46
Sorted entries 28= 1104 Ascii = 99
Sorted entries 29= 1154 Ascii = 111
Sorted entries 30= 1234 Ascii = 109
Sorted entries 31= 1515 Ascii = 114
Sorted entries 32= 1649 Ascii = 110
Sorted entries 33= 1722 Ascii = 108
Sorted entries 34= 2144 Ascii = 97
Sorted entries 35= 2204 Ascii = 116
Sorted entries 36= 2352 Ascii = 115
Sorted entries 37= 2622 Ascii = 117
Sorted entries 38= 2696 Ascii = 105
Sorted entries 39= 3266 Ascii = 101
Sorted entries 40= 4999 Ascii = 32
```

4) Created the priority queue to store these Huffman nodes with lease frequencies at first. Note: I explored the ways of using Radix sort and Count Sort to optimize Huffman Tree creation. There are papers published which states effectively proper use of sorting and using it again and again optimizes encoding. In case of priority queue creation, sorting could be avoided.
5) After creation of Priority queue, the Huffman node array was freed.
6) Create a tree following Greedy algorithm. Start with two nodes with least frequencies. Add their frequencies, create an intermediate node and add to priority queue. Note: By default, priority queue inserts in higher frequency order, overload the Compare method to do the opposite.
7) Store the Huffman Encoding table in the Map. Ascii Key is the key and its second is string of binary data symbol.
8) Create header of Encoding Map.

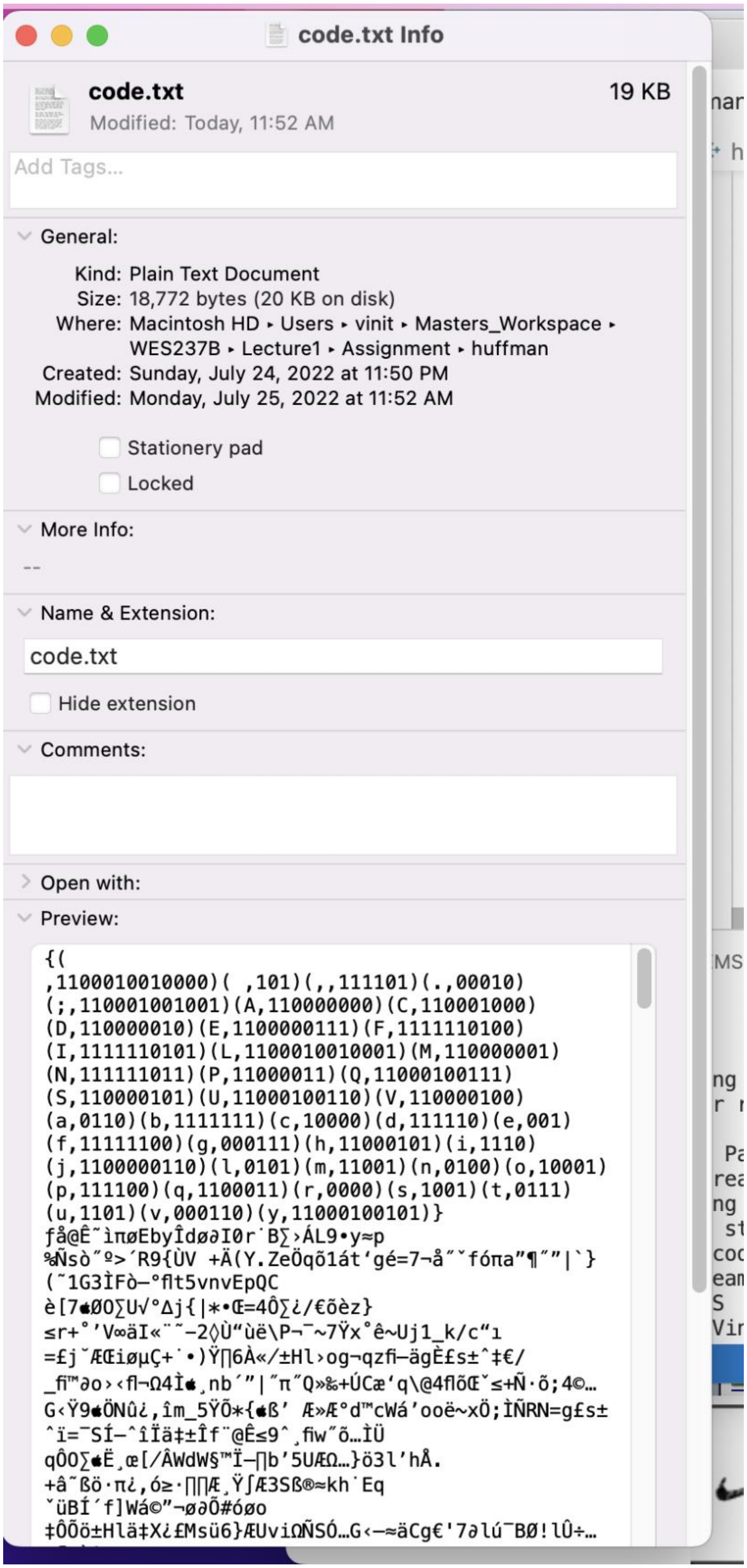
```
Ascii value  ---  Symbol  --  Code
10
            1100010010000
32                         101
44            ,            111101
46            .            00010
59            ;            110001001001
65            A             110000000
```

67	C	110001000
68	D	110000010
69	E	1100000111
70	F	1111110100
73	I	1111110101
76	L	1100010010001
77	M	110000001
78	N	111111011
80	P	11000011
81	Q	11000100111
83	S	110000101
85	U	11000100110
86	V	110000100
97	a	0110
98	b	1111111
99	c	10000
100	d	111110
101	e	001
102	f	11111100
103	g	000111
104	h	11000101
105	i	1110
106	j	1100000110
108	l	0101
109	m	11001
110	n	0100
111	o	10001
112	p	111100
113	q	1100011
114	r	0000
115	s	1001
116	t	0111
117	u	1101
118	v	000110
121	y	11000100101

9) Start reading input stream character by character, use map to find out the symbol. Add it in string object.
10) Write header created to the unsigned char type vector. (1 Byte per entry). Header format is
    {'invalid bit'(Ascii Key, "Binary symbol")(…)}
11) Use string object to read 8 Byte of bitstream data. Use 1 Byte of unsigned char data type to set/unset corresponding bits. Store the unsigned char to vector created for header. Make sure, padding is taken care of. In case the size of file is not multiple of 8, then

there would be invalid byte information would be saved. Header has this information of invalid bytes. 1st position of header. The encoded string updates this information.

12) Get the size of vector, dynamically allocate the memory and populate the coded entries.
13) Store the file in code.txt



code.txt 19 KB

Modified: Today, 11:52 AM

General:

Kind: Plain Text Document
Size: 18,772 bytes (20 KB on disk)
Where: Macintosh HD ▸ Users ▸ vinit ▸ Masters_Workspace ▸ WES237B ▸ Lecture1 ▸ Assignment ▸ huffman
Created: Sunday, July 24, 2022 at 11:50 PM
Modified: Monday, July 25, 2022 at 11:52 AM

□ Stationery pad
□ Locked

More Info:

--

Name & Extension:

code.txt

□ Hide extension

Comments:

Preview:

```
{(
,1100010010000)( ,101)(,,111101)(.,00010)
(;,110001001001)(A,110000000)(C,110001000)
(D,110000010)(E,1100000111)(F,1111110100)
(I,1111110101)(L,1100010010001)(M,110000001)
(N,111111011)(P,11000011)(Q,11000100111)
(S,110000101)(U,11000100110)(V,110000100)
(a,0110)(b,1111111)(c,10000)(d,111110)(e,001)
(f,11111100)(g,000111)(h,11000101)(i,1110)
(j,1100000110)(l,0101)(m,11001)(n,0100)(o,10001)
(p,111100)(q,1100011)(r,0000)(s,1001)(t,0111)
(u,1101)(v,000110)(y,11000100101)}
få@Ê˜ìπøEbyÎdø∂I0r˙B∑›ÁL9•y≈p
‰Ñsò˝º>´R9{ÙV +Ä(Y.ZeÖqõ1át‘gé=7¬å˝ˇfóπa”¶˝”|`}
(˜1G3ÌFò–°flt5vnvEpQC
è[7ØO∑U√°∆j{|∗•Œ=4Ô∑¿/€õèz}
≤r+˚’V∞äI«¨˜–2◊Ù“ùë\P¬¯~7Ÿx˚ê~Uj1_k/c“ı
=£j˘ÆŒiøµÇ+˙•)Ÿ∏6À«/±Hl›og¬qzfi–ägÈ£s±ˆ‡€/
_fi™∂o›‹fl¬Ω4Ì¸nb´”|˝π˝Q»‰+ÚCæ‘q\@4flõŒˇ≤+Ñ·õ;4©…
G‹Ÿ9ÖNû¿,îm_5ŸÕ∗{ß’ Æ»Æ°d™cWá’ooë~xÖ;ÌÑRN=g£s±
ˆï=¯SÍ–ˆîÏä‡±Îf¨@Ê≤9ˆ¸fiw˝õ…ÌÜ
qÔO∑Ë¸œ[/ÂWdW§™Ï–∏b’5UÆΩ…}ö3l’hÅ.
+â˜ßö·π¿,ó≥·∏∏Æ¸Ÿ∫Æ3Sß®≈kh˙Eq
ˇüBÍ´f]Wá©”¬ø∂Õ#óøo
‡ÔÕö±Hlä‡X¿£Msü6}ÆUviΩÑSÓ…G‹–≈äCg€'7∂lú¯BØ!lÛ÷…
```

In the preview section the file shows , Header followed by binary data. The file size would relatively remain unaffected due to Header. Improvement could be made in storing the information. For a larger file, the header would not impact the compression much. So for the input file given in the assignment the compressed size was 18KB from 37 KB

Huffman Decoding:

1) Read the data from the code.txt. Parse the header and create MAP.

```
*********  Decode   *******
Header Start
Header creation successful
Ascii value  ---  Symbol  --  Code
10
           1100010010000
32                      101
44          ,           111101
46          .           00010
59          ;           110001001001
65          A            110000000
67          C            110001000
68          D            110000010
69          E            1100000111
70          F            1111110100
73          I           1111110101
76          L            1100010010001
77          M             110000001
78          N            111111011
80          P            11000011
81          Q             11000100111
83          S            110000101
85          U            11000100110
86          V            110000100
97          a            0110
98          b            1111111
99          c            10000
100          d            111110
101          e            001
102          f            11111100
103          g            000111
104          h            11000101
105          i            1110
```

```
106         j           1100000110
108         l           0101
109         m           11001
110         n           0100
111         o           10001
112         p           111100
113         q           1100011
114         r           0000
115         s           1001
116         t           0111
117         u           1101
118         v           000110
121         y           11000100101
Creating Decoder Tree , node address = 0x127705f50
```

2) Create the Huffman Tree by parsing through every entry iterating over the map.
3) Read the binary data and create a string object to store intermediate bit streams information. Take care of invalid bits. This information is obtained from header.
4) Decode by iterating over the tree. Keep traversing the tree till leaf is reached. From the encoding table, fetch the binary symbol and store is in vector of unsigned char.
5) Allocate the memory dynamically and store the decoded bit streams in unsigned char memory location.
6) For properly decoded bit streams, the output.txt file is created.
7) Diff command compares the file input.txt and output.txt and outputs the result.

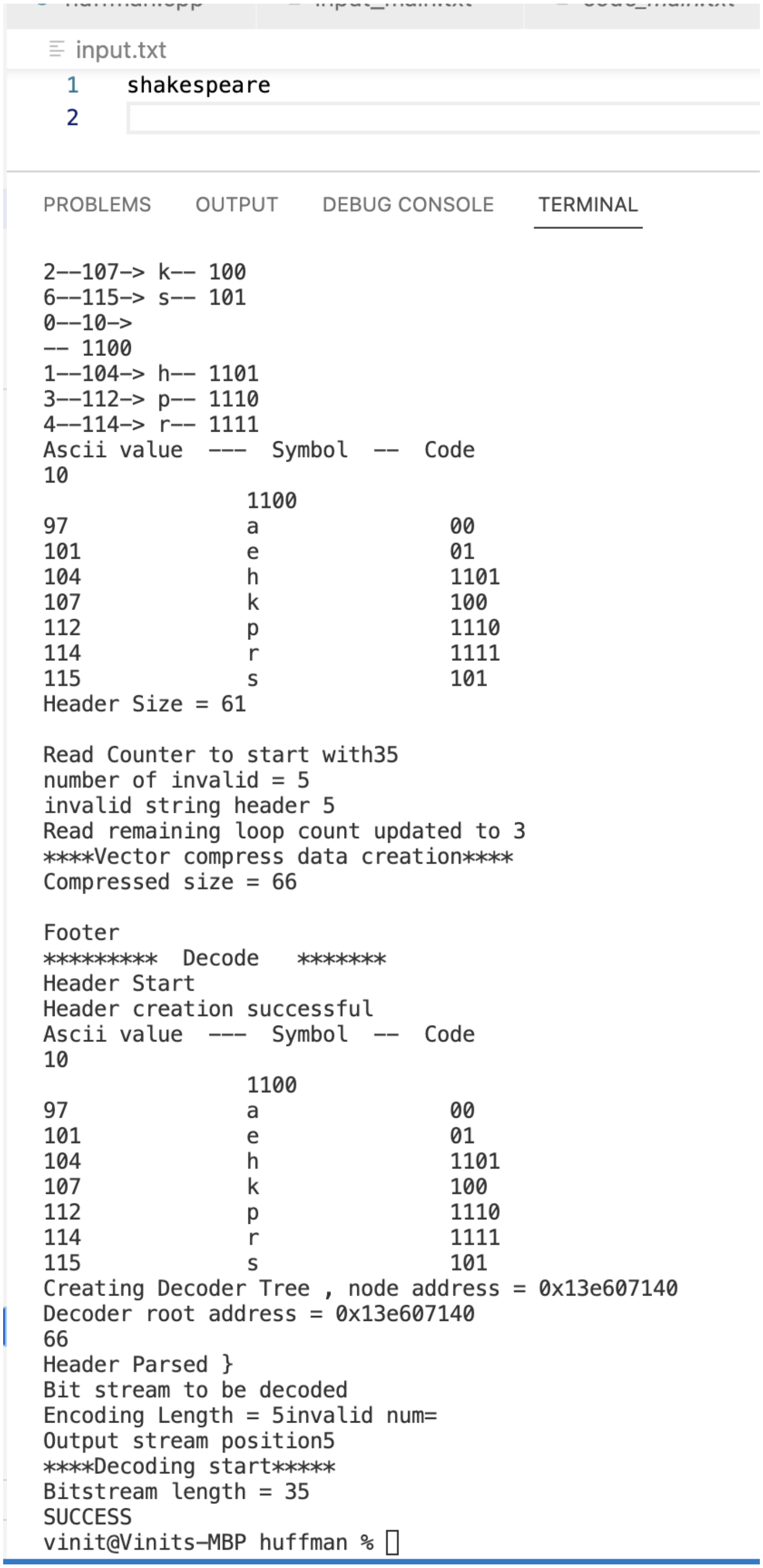
Output result of input.txt of assignment

```
****Decoding start*****
Bitstream length = 146435
SUCCESS
Original file of 37KB is now of 20 KB
```

How to compile:

1)Go to GITHUB link: https://github.com/VinitSaah/WES237B.git
2) Clone Assignment1
3) Run the make file : 1) make clean 2) make
4 Run: ./CODEC input.txt code.txt output.txt

SAMPLE OUTPUT FOR Shakespeare

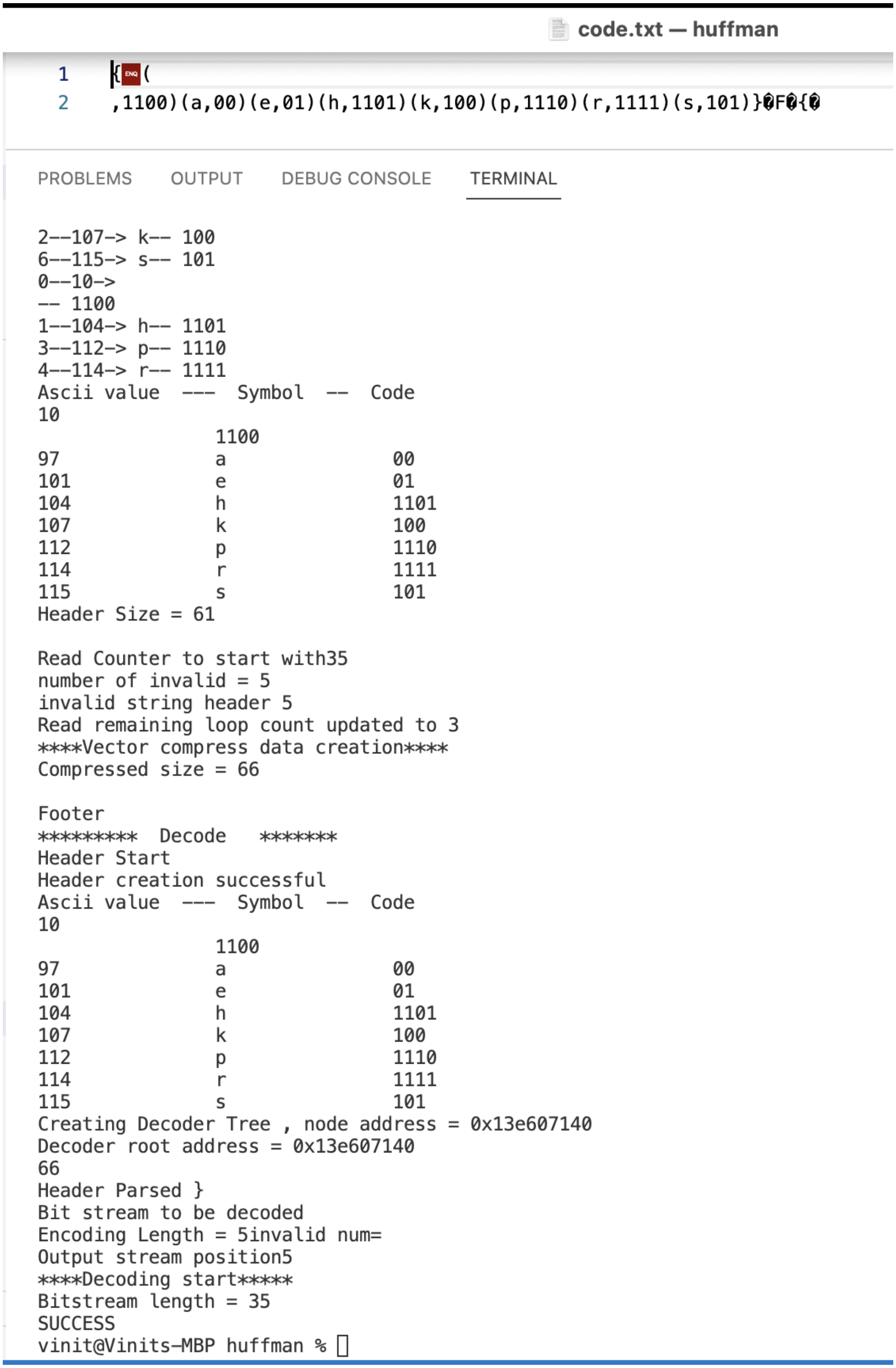


```
shakespeare
```

```
2--107-> k-- 100
6--115-> s-- 101
0--10->
-- 1100
1--104-> h-- 1101
3--112-> p-- 1110
4--114-> r-- 1111
Ascii value  ---  Symbol  --  Code
10
                1100
97              a               00
101             e               01
104             h               1101
107             k               100
112             p               1110
114             r               1111
115             s               101
Header Size = 61

Read Counter to start with35
number of invalid = 5
invalid string header 5
Read remaining loop count updated to 3
****Vector compress data creation****
Compressed size = 66

Footer
*********  Decode   *******
Header Start
Header creation successful
Ascii value  ---  Symbol  --  Code
10
                1100
97              a               00
101             e               01
104             h               1101
107             k               100
112             p               1110
114             r               1111
115             s               101
Creating Decoder Tree , node address = 0x13e607140
Decoder root address = 0x13e607140
66
Header Parsed }
Bit stream to be decoded
Encoding Length = 5invalid num=
Output stream position5
****Decoding start*****
Bitstream length = 35
SUCCESS
vinit@Vinits-MBP huffman % 
```

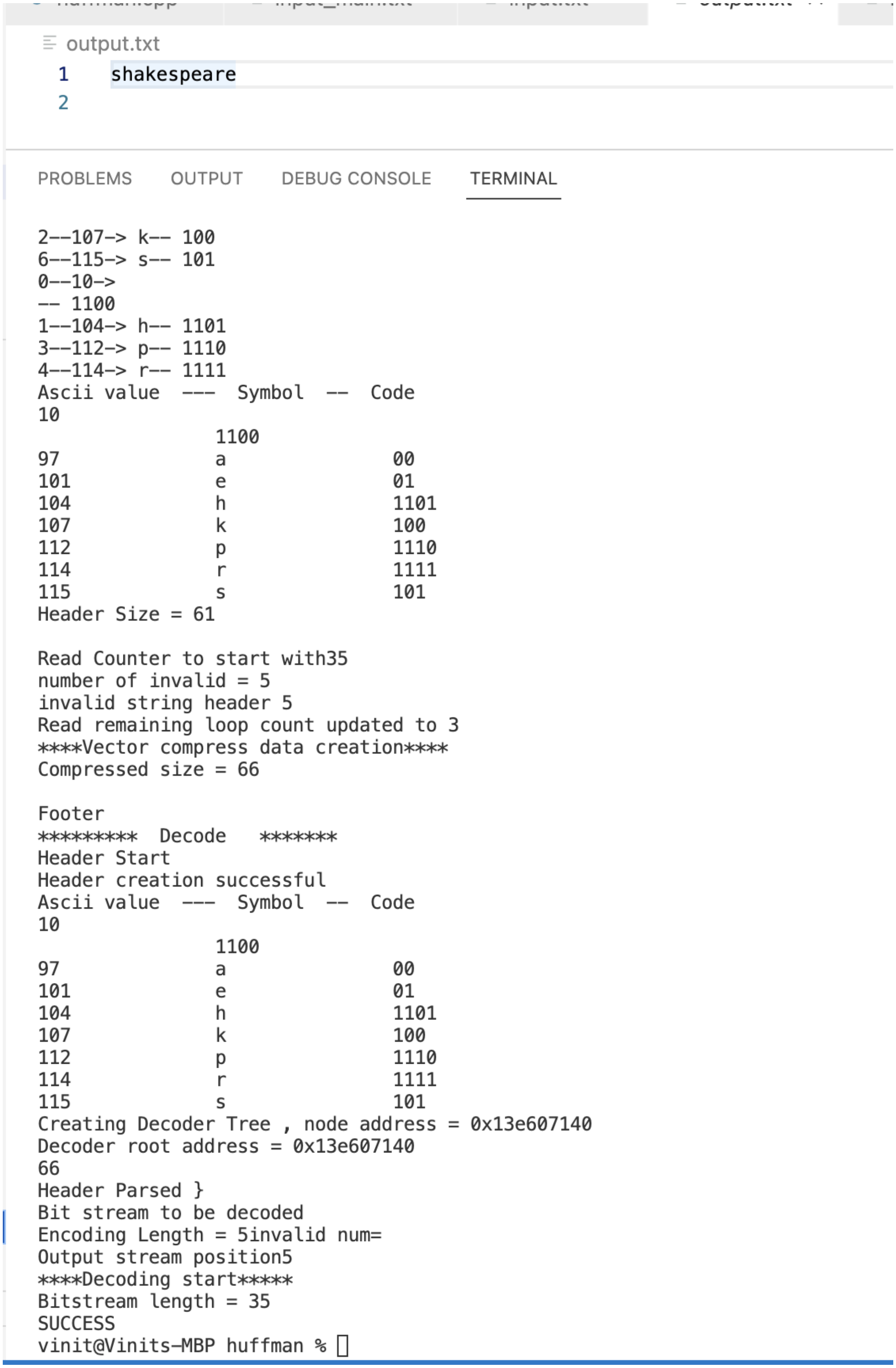


```
{ENQ(
,1100)(a,00)(e,01)(h,1101)(k,100)(p,1110)(r,1111)(s,101)}�F�{�
```

```
2--107-> k-- 100
6--115-> s-- 101
0--10->
-- 1100
1--104-> h-- 1101
3--112-> p-- 1110
4--114-> r-- 1111
Ascii value  ---  Symbol  --  Code
10
                1100
97              a               00
101             e               01
104             h               1101
107             k               100
112             p               1110
114             r               1111
115             s               101
Header Size = 61

Read Counter to start with35
number of invalid = 5
invalid string header 5
Read remaining loop count updated to 3
****Vector compress data creation****
Compressed size = 66

Footer
*********  Decode   *******
Header Start
Header creation successful
Ascii value  ---  Symbol  --  Code
10
                1100
97              a               00
101             e               01
104             h               1101
107             k               100
112             p               1110
114             r               1111
115             s               101
Creating Decoder Tree , node address = 0x13e607140
Decoder root address = 0x13e607140
66
Header Parsed }
Bit stream to be decoded
Encoding Length = 5invalid num=
Output stream position5
****Decoding start*****
Bitstream length = 35
SUCCESS
vinit@Vinits-MBP huffman % 
```



```
shakespeare
```

```
2--107-> k-- 100
6--115-> s-- 101
0--10->
-- 1100
1--104-> h-- 1101
3--112-> p-- 1110
4--114-> r-- 1111
Ascii value  ---  Symbol  --  Code
10
                1100
97              a               00
101             e               01
104             h               1101
107             k               100
112             p               1110
114             r               1111
115             s               101
Header Size = 61

Read Counter to start with35
number of invalid = 5
invalid string header 5
Read remaining loop count updated to 3
****Vector compress data creation****
Compressed size = 66

Footer
*********  Decode   *******
Header Start
Header creation successful
Ascii value  ---  Symbol  --  Code
10
                1100
97              a               00
101             e               01
104             h               1101
107             k               100
112             p               1110
114             r               1111
115             s               101
Creating Decoder Tree , node address = 0x13e607140
Decoder root address = 0x13e607140
66
Header Parsed }
Bit stream to be decoded
Encoding Length = 5invalid num=
Output stream position5
****Decoding start*****
Bitstream length = 35
SUCCESS
vinit@Vinits-MBP huffman % 
```