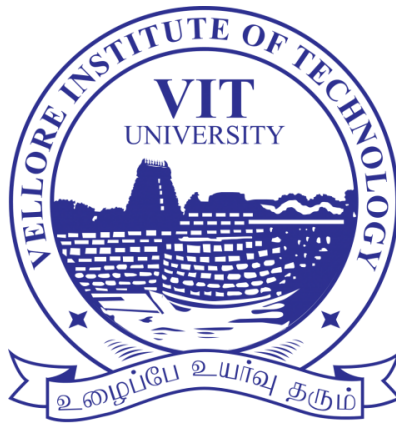


IMPLEMENTATION OF HUFFMAN CODING

Submitted by
SHAURYA CHOUDHARY
(18BCE2113)

Under the guidance of
PROF. SHALINI L




Vellore Institute of Technology, Vellore
Tamil Nadu – 632014

CONTENTS

 *Abstract*

 *Introduction to the problem*

 *Related works*

 *Motivation on the project*

 *Proposed work*

 *Experimentation*

 *Conclusion*

 *References*

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my teacher Prof. Shalini L who gave me the golden opportunity to do this wonderful project on Data Structures, which also helped me in doing a lot of Research and I came up with an idea of demonstrating lossless data compression by the “Implementation of Huffman coding” which helped me realize the real life use of Data Structures and Algorithm and I came to know about so many new things, I am really thankful to them.

Secondly, I would also like to thank my friends who helped me in making this project possible and a lot in finalizing this project within the limited time frame.

DATE: 02/04/19

SHAURYA CHOUDHARY
(18BCE2113)

ABSTRACT

Huffman Coding is an approach to text compression originally developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes". In computer science and information theory, it is one of many lossless data compression algorithms. It is a statistical compression method that converts characters into variable length bit strings and produces a prefix code. Most-frequently occurring characters are converted to shortest bit strings; least frequent, the longest.

The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

INTRODUCTION TO THE PROBLEM

Given

A set of symbols and their weights (usually proportional to probabilities or equal to their frequencies).

Find

A prefix-free binary code (a set of code words) with minimum expected code word length (equivalently, a tree with minimum weighted path length from the root).

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

Step 1:- Create a leaf node for each symbol and add it to the priority queue (i.e. Create a min heap of Binary trees and heapify it).

Step 2:- While there is more than one node in the queue (i.e. min heap):

- Remove the two nodes of highest priority (lowest probability or lowest frequency) from the queue.
- Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities (frequencies).
- Add the new node to the queue.

Step 3:- The remaining node is the root node and the Huffman tree is complete.

Joining trees by frequency is the same as merging sequences by length in optimal merge.

Since a node with only one child is not optimal, any Huffman coding corresponds to a full binary tree.

RELATED WORKS

Arithmetic coding can be viewed as a generalization of Huffman coding; indeed, in practice arithmetic coding is often preceded by Huffman coding, as it is easier to find an arithmetic code for a binary input than for a non-binary input. Also, although arithmetic coding offers better compression performance than Huffman coding, Huffman coding is still in wide use because of its simplicity, high speed and lack of encumbrance by patents.

Huffman coding today is often used as a "back-end" to some other compression method. DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by Huffman coding.

MOTIVATION ON THE PROJECT

Huge data system applications require storage of large volumes of data set, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, the proliferation of communication networks is resulting in massive transfer of data over communication links. Compressing data to be stored or transmitted reduces storage and communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Here efficient method for decoding the compressed data is proposed. This paper aims toward the implementation of a high speed Huffman decoding system. This proposed model enhances the speed of decoding operation.

PROPOSED WORK

➤ INTRODUCTION TO HUFFMAN CODING:

Let us suppose, we need to store a string of length 1000 that comprises characters a, e, n, and z. To storing it as 1-byte characters will require 1000 byte (or 8000 bits) of space. If the symbols in the string are encoded as (00=a, 01=e, 10=n, 11=z), then the 1000 symbols can be stored in 2000 bits saving 6000 bits of memory.

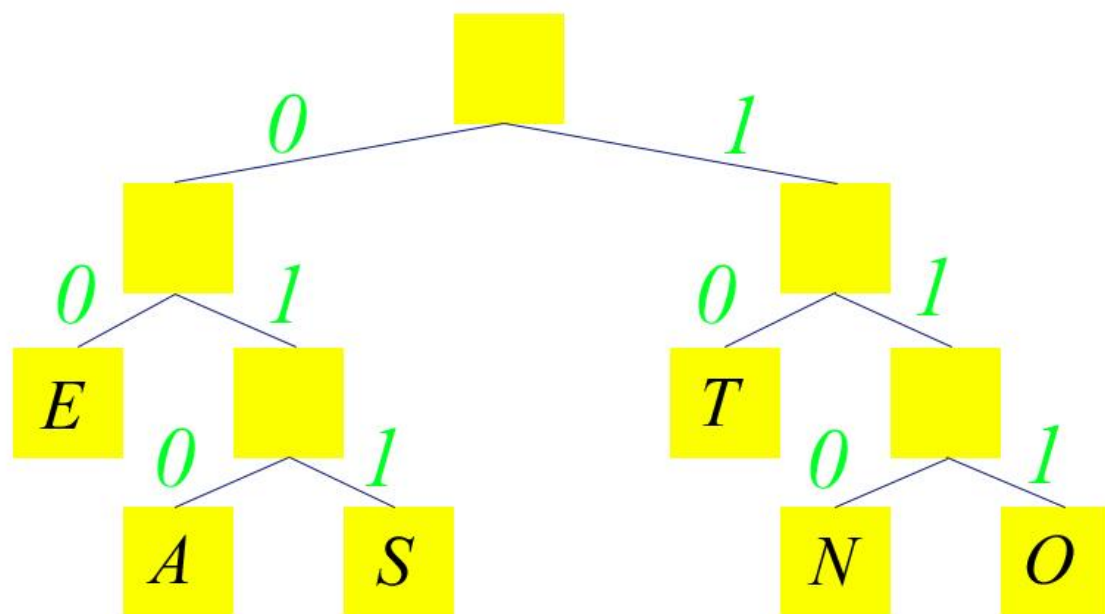
The number of occurrence of a symbol in a string is called its frequency. When there is considerable difference in the frequencies of different symbols in a string, variable length codes can be assigned to the symbols based on their relative frequencies. The most common characters can be represented using shorter codes than are used for less common source symbols. More is the variation in the relative frequencies of symbols, it is more advantageous to use variable length codes for reducing the size of coded string.

Since the codes are of variable length, it is necessary that no code is a prefix of another so that the codes can be properly decode. Such codes are called prefix code (sometimes called "prefix-free codes", that is, the code representing some particular symbol is never a prefix of the code representing any other symbol). Huffman coding is so much widely used for creating prefix codes that the term "Huffman code" is sometimes used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Huffman was able to design the most efficient compression method of this type: no other mapping of individual source symbols to unique strings of bits (i.e. codes) will require lesser space for storing a piece of text when the actual symbol frequencies agree with those used to create the code.

➤ **BASIC TECHNIQUE:**

In Huffman Coding, the complete set of codes can be represented as a binary tree, known as a Huffman tree. This Huffman tree is also a coding tree i.e. a full binary tree in which each leaf is an encoded symbol and the path from the root to a leaf is its code word. By convention, bit '0' represents following the left child and bit '1' represents following the right child. One code bit represents each level. Thus more frequent characters are near the root and are coded with few bits, and rare characters are far from the root and are coded with many bits.



HUFFMAN TREE

First of all, the source symbols along with their frequencies of occurrence are stored as leaf nodes in a regular array, the size of which depends on the number of symbols, n . A finished tree has up to n leaf nodes and $n - 1$ internal nodes.

➤ ***OPTIMAL MERGE:***

Let $D = \{n_1, \dots, n_k\}$ be the set of lengths of sequences to be merged. Take the two shortest sequences, $n_i, n_j \in D$, such that $n \geq n_i$ and $n \geq n_j \forall n \in D$. Merge these two sequences. The new set D is $D' = (D - \{n_i, n_j\}) \cup \{n_i + n_j\}$. Repeat until there is only one sequence.

Since efficient priority queue data structures require $O(\log n)$ time per insertion, and a tree with n leaves has $2n-1$ nodes, this algorithm operates in $O(n \log n)$ time.

The worst case for Huffman coding (or, equivalently, the longest Huffman coding for a set of characters) is when the distribution of frequencies follows the Fibonacci numbers.

If the estimated probabilities of occurrence of all the symbols are same and the number of symbols are a power of two, Huffman coding is same as simple binary block encoding, e.g., ASCII coding.

Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e. a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown, not identically distributed, or not independent (e.g., "cat" is more common than "cta").

EXPERIMENTATION

➤ **C++ PROGRAM CODE:**

```
// Name: SHAURYA CHOUDHARY
// Reg. No.: 18BCE2113
// Course: DATA STRUCTURES AND ALGORITHM
// Faculty: PROF. SHALINI L
```

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
struct node
{
    char info;
    int freq;
    char *code;
    node *Llink;
    node *Rlink;
};
```

```
class BinaryTree // Coding Tree
{
private:
    node *root;
public:
    BinaryTree() { root=NULL; }
    void print();

    void assign_code(int i);
    void print_code(char c);
```

```

        void encode(const char str[]);
        void print_symbol(char cd[], int &f, int length);
        void decode(char cd[], int size);

friend class minHeap;
friend class HuffmanCode;
};

class minHeap
{
private:
    BinaryTree *T; // Array of Binary Trees
    int n;
public:
    minHeap();
    void heapify(int i);
    BinaryTree remove();
    void insert(BinaryTree b);
    void print();
    friend class HuffmanCode;
};

class HuffmanCode
{
private:
    BinaryTree HuffmanTree;//A Huffman Tree with symbols as external nodes.
public:
    HuffmanCode();
};

HuffmanCode::HuffmanCode()
{
    minHeap Heap;
    while (Heap.T[0].root->freq>1)
    {
        BinaryTree l=Heap.remove();

```

```

        cout<<"\nAfter removing "<<l.root->freq<<endl;
        Heap.print();
        BinaryTree r=Heap.remove();
        cout<<"\nAfter removing "<<r.root->freq<<endl;
        Heap.print();
        HuffmanTree.root=new node;
        HuffmanTree.root->info='\0';
        HuffmanTree.root->freq=l.root->freq + r.root->freq;
        HuffmanTree.root->Llink=l.root;
        HuffmanTree.root->Rlink=r.root;
        Heap.insert(HuffmanTree);

        cout<<"\nAfter inserting "<<l.root->freq<<"+ "<<r.root->freq<< "=
"<<HuffmanTree.root->freq<<endl;

        Heap.print();
    }

```

```

    cout<<"\nThe process is completed and Huffman Tree is obtained\n";
    system ("pause");
    HuffmanTree=Heap.T[1];// This tree is our HuffmanTree used for coding
    delete []Heap.T;

    cout<<"Traversal of Huffman Tree\n\n";
    HuffmanTree.print();
    system ("pause");

    cout<<"\nThe symbols with their codes are as follows\n";
    HuffmanTree.assign_code(0);
    system ("pause");// Codes are assigned to the symbols

    cout<<"Enter the string to be encoded by Huffman Coding: ";
    char *str;
    str=new char[50];
    cin>>str;
    HuffmanTree.encode(str);
    system ("pause");

    int length;

    cout<<"Enter the code to be decoded by Huffman Coding: ";
    char *cd;
    cd=new char[60];

```

```

        cin>>cd;

        cout<<"Enter its code length: ";
        cin>>length;

        HuffmanTree.decode(cd,length);

        system ("pause");
    }

minHeap::minHeap()
{
    cout<<"Enter no. of symbols:";
    cin>>n;

    T= new BinaryTree [n+1];
    T[0].root=new node;
    T[0].root->freq=n; //Number of elements in min. Heap is stored in the zeroth element of the
heap
    for (int i=1; i<=n; i++)
    {
        T[i].root=new node;

        cout<<"Enter characters of string :- ";
        cin>>T[i].root->info;
        cout<<"and their frequency of occurrence in the string:- ";
        cin>>T[i].root->freq;
        T[i].root->code=NULL;
        T[i].root->Llink=NULL;
        T[i].root->Rlink=NULL;

    }

    cout<<endl;
    int i=(int)(n / 2);
    cout<<"\nAs elements are entered\n";
    print();
    while (i>0)
    {
        heapify(i);
        i--;
    }
}

```

```

    cout<<"\nAfter heapification \n";
    print();
}

int min(node *a, node *b)
{ if (a->freq <= b->freq) return a->freq;          else return b->freq;}

void swap(BinaryTree &a, BinaryTree &b)
{ BinaryTree c=a;          a=b;          b=c;}

void minHeap::heapify(int i)
{
    while(1)
    {
        if (2*i > T[0].root->freq)
            return;
        if (2*i+1 > T[0].root->freq)
        {
            if (T[2*i].root->freq <= T[i].root->freq)
                swap(T[2*i],T[i]);
            return;
        }
        int m=min(T[2*i].root,T[2*i+1].root);
        if (T[i].root->freq <= m)
            return;
        if (T[2*i].root->freq <= T[2*i+1].root->freq)
        {
            swap(T[2*i],T[i]);
            i=2*i;
        }
        else
        {
            swap(T[2*i+1],T[i]);
            i=2*i+1;
        }
    }
}

```

```
}
```

```
BinaryTree minHeap::remove()
```

```
{
```

```
    BinaryTree b=T[1];
```

```
    T[1]= T[T[0].root->freq];
```

```
    T[0].root->freq--;
```

```
    if (T[0].root->freq!=1)
```

```
        heapify(1);
```

```
    return b;
```

```
}
```

```
void minHeap::insert(BinaryTree b)
```

```
{
```

```
    T[0].root->freq++;
```

```
    T[T[0].root->freq]=b;
```

```
    int i=(int) (T[0].root->freq /2 );
```

```
    while (i>0)
```

```
    {
```

```
        heapify (i);
```

```
        i=(int) (i /2 );
```

```
    }
```

```
}
```

```
int isleaf(node *nd)
```

```
{ if(nd->info=="\0") return 0; else return 1;}
```

```
void BinaryTree::assign_code(int i)
```

```
{
```

```
    if (root==NULL)
```

```
        return;
```

```
    if (isleaf(root))
```

```
    {
```

```
        root->code[i]='\0';
```

```
        cout<<root->info<<"\t"<<root->code<<"\n";
```

```
        return;
```

```

    }

    BinaryTree l,r;

    l.root=root->Llink;

    r.root=root->Rlink;

    l.root->code=new char[i+1];
    r.root->code=new char[i+1];

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

        l.root->code[k]=root->code[k];
        r.root->code[k]=root->code[k];

    }

    l.root->code[i]='0';
    r.root->code[i]='1';

    i++;

    l.assign_code(i);
    r.assign_code(i);
}

void BinaryTree::encode(const char str[])
{
    if (root==NULL)
return;

    int i=0;

    cout<<"Encoded code for the input string "<<str<<" is\n";
    while (1)
    {

        if (str[i]=='\0')
        {

            cout<<endl;

            return;

        }

        print_code(str[i]);

        i++;

    }
}

```



```

void BinaryTree::print_code(char c)
{
    int f=0;
    if (isleaf(root))
    {
        if (c==root->info)
        {
            f=1;
            cout<<root->code;
        }
        return ;
    }
    BinaryTree l,r;
    l.root=root->Llink;
    if (f!=1)
    l.print_code(c);
    r.root=root->Rlink;
    if (f!=1)
    r.print_code(c);
}

int isequal(const char a[], const char b[], int length)
{
    int i=0;
    while (i<length)
    {
        if(b[i]!=a[i])
            return 0;
        i++;
    }
    if (a[i]!='\0')
        return 0;
    return 1;
}

void BinaryTree::decode(char cd[], int size)

```

```

{
    if (root==NULL)
return;

    int i=0;
    int length=0;
    int f;
    char *s;
    cout<<"Decoded string for the input code "<<cd<<" is\n";
    while (i<size)
    {
        f=0;
        s=&cd[i];
        while (f==0)
        {
            length++;
            print_symbol(s,f,length);
        }
        i=i+length;
        length=0;
    }
    cout<<endl;
}

```

```

void BinaryTree::print_symbol(char cd[], int &f, int length)

```

```

{
    if (isleaf(root))
    {
        if (isequal(root->code, cd, length))
        {
            f=1;
            cout<<root->info;
        }
        return;
    }

    BinaryTree l,r;
    l.root=root->Llink;

```

```

        if (f!=1)
        l.print_symbol(cd,f,length);
        r.root=root->Rlink;
        if (f!=1)
        r.print_symbol(cd,f,length);
    }

```

```

void BinaryTree::print()
{
    if (root==NULL)
        return;
    cout<<root->info<<"\t"<<root->freq<<"\n";
    if (isleaf(root))
        return;
    BinaryTree l,r;
    l.root=root->Llink;
    r.root=root->Rlink;
    l.print();
    r.print();
}

```

```

int power(int i, int j)
{
    int n=1;
    for (int k=1; k<=j; k++)
        n=n*i;
    return n;
}

```

```

int ispowerof2(int i)
{
    if (i==1)
        return 0;
    if (i==0)
        return 1;
    while (i>2)

```

```

        {
            if (i%2!=0)
                return 0;
            i=i/2;
        }
        return 1;
    }

int fn(int l)
{
    if (l==1||l==0)
        return 0;
    return 2*fn(l-1)+1;
}

void minHeap::print()
{
    cout<<"The Heap showing the root frequencies of the Binary Trees are:\n";
    if (T[0].root->freq==0)
    {
        cout<<endl;
        system ("pause");
        return;
    }
    int level=1;
    while( T[0].root->freq >= power(2,level) )
        level++;
    if(level==1)
    {
        cout<<T[1].root->freq<<"\n";
        system ("pause");
        return;
    }
    for (int i=1; i<=T[0].root->freq; i++)
    {
        if (ispowerof2(i))

```

```
        {cout<<"\n"; level--;}
        for (int k=1; k<=fn(level); k++)
            cout<<" ";
        cout<<T[i].root->freq<<" ";
        for (int k=1; k<=fn(level); k++)
            cout<<" ";

    }

    cout<<endl;
    system ("pause");
}

int main()
{
    HuffmanCode c;

    system ("pause");
    return 0;
}
```

➤ **OUTPUT:**

```
C:\Users\shaun\Desktop\DSA Project\188CE2113_DSAProject.exe
Enter no. of symbols:8
Enter characters of string :- a
and their frequency of occurrence in the string:- 18
Enter characters of string :- b
and their frequency of occurrence in the string:- 2
Enter characters of string :- c
and their frequency of occurrence in the string:- 7
Enter characters of string :- d
and their frequency of occurrence in the string:- 1
Enter characters of string :- e
and their frequency of occurrence in the string:- 27
Enter characters of string :- f
and their frequency of occurrence in the string:- 13
Enter characters of string :- g
and their frequency of occurrence in the string:- 9
Enter characters of string :- h
and their frequency of occurrence in the string:- 11

As elements are entered
The Heap showing the root frequencies of the Binary Trees are:
      18
     2   7
    1  27 13  9
   11
Press any key to continue . . .
```

```
C:\Users\shaun\Desktop\DSA Project\188CE2113_DSAProject.exe

As elements are entered
The Heap showing the root frequencies of the Binary Trees are:
      18
     2   7
    1  27 13  9
   11
Press any key to continue . . .

After heapification
The Heap showing the root frequencies of the Binary Trees are:
      1
     2   7
    11  27 13  9
   18
Press any key to continue . . .

After removing 1
The Heap showing the root frequencies of the Binary Trees are:
      2
     11  7
    18 27 13  9
Press any key to continue . . .

After removing 2
The Heap showing the root frequencies of the Binary Trees are:
      7
     11  9
    18 27 13
Press any key to continue . . .
```

```
C:\Users\shaun\Desktop\DSA Project\188CE2113_DSAProject.exe
18 27 13 9
Press any key to continue . . .

After removing 2
The Heap showing the root frequencies of the Binary Trees are:
  7
11  9
18 27 13
Press any key to continue . . .

After inserting 1+2= 3
The Heap showing the root frequencies of the Binary Trees are:
  3
11  7
18 27 13 9
Press any key to continue . . .

After removing 3
The Heap showing the root frequencies of the Binary Trees are:
  7
11  9
18 27 13
Press any key to continue . . .

After removing 7
The Heap showing the root frequencies of the Binary Trees are:
  9
11 13
18 27
Press any key to continue . . .
```

```
C:\Users\shaun\Desktop\DSA Project\188CE2113_DSAProject.exe
18 27 13
Press any key to continue . . .

After removing 7
The Heap showing the root frequencies of the Binary Trees are:
  9
11 13
18 27
Press any key to continue . . .

After inserting 3+7= 10
The Heap showing the root frequencies of the Binary Trees are:
  9
11 10
18 27 13
Press any key to continue . . .

After removing 9
The Heap showing the root frequencies of the Binary Trees are:
 10
11 13
18 27
Press any key to continue . . .

After removing 10
The Heap showing the root frequencies of the Binary Trees are:
 11
18 13
27
Press any key to continue . . .
```

```
C:\Users\shaur\Desktop\DSA Project\188CE2113_DSAProject.exe
11 13
18 27
Press any key to continue . . .

After removing 10
The Heap showing the root frequencies of the Binary Trees are:
11
18 13
27
Press any key to continue . . .

After inserting 9+10= 19
The Heap showing the root frequencies of the Binary Trees are:
11
18 13
27 19
Press any key to continue . . .

After removing 11
The Heap showing the root frequencies of the Binary Trees are:
13
18 19
27
Press any key to continue . . .

After removing 13
The Heap showing the root frequencies of the Binary Trees are:
18
27 19
Press any key to continue . . .
```

```
C:\Users\shaur\Desktop\DSA Project\188CE2113_DSAProject.exe
After removing 13
The Heap showing the root frequencies of the Binary Trees are:
18
27 19
Press any key to continue . . .

After inserting 11+13= 24
The Heap showing the root frequencies of the Binary Trees are:
18
24 19
27
Press any key to continue . . .

After removing 18
The Heap showing the root frequencies of the Binary Trees are:
19
24 27
Press any key to continue . . .

After removing 19
The Heap showing the root frequencies of the Binary Trees are:
24
27
Press any key to continue . . .

After inserting 18+19= 37
The Heap showing the root frequencies of the Binary Trees are:
24
27 37
Press any key to continue . . .
```



```
C:\Users\shaur\Desktop\DSA Project\188CE2113_DSAProject.exe
24
27 37
Press any key to continue . . .

After removing 24
The Heap showing the root frequencies of the Binary Trees are:
27
37
Press any key to continue . . .

After removing 27
The Heap showing the root frequencies of the Binary Trees are:
37
Press any key to continue . . .

After inserting 24+27= 51
The Heap showing the root frequencies of the Binary Trees are:
37
51
Press any key to continue . . .

After removing 37
The Heap showing the root frequencies of the Binary Trees are:
51
Press any key to continue . . .

After removing 51
The Heap showing the root frequencies of the Binary Trees are:
Press any key to continue . . .
```

```
C:\Users\shaur\Desktop\DSA Project\188CE2113_DSAProject.exe
After removing 51
The Heap showing the root frequencies of the Binary Trees are:
Press any key to continue . . .

After inserting 37+51= 88
The Heap showing the root frequencies of the Binary Trees are:
88
Press any key to continue . . .

The process is completed and Huffman Tree is obtained
Press any key to continue . . .
Traversal of Huffman Tree

      88
     37
a    18
    19
g    9
    10
     3
d    1
b    2
c    7
    51
    24
h    11
f    13
e    27
Press any key to continue . . .
```

```
C:\Users\shaur\Desktop\DSA Project\18BCE2113_DSAProject.exe
Press any key to continue . . .
Traversal of Huffman Tree

      88
     37
    18
    19
   9
  10
  3
 1
 2
 7
51
24
11
13
27
Press any key to continue . . .

The symbols with their codes are as follows
a      00
g      010
d      01100
b      01101
c      0111
h      100
f      101
e      11
Press any key to continue . . .
```

```
C:\Users\shaur\Desktop\DSA Project\18BCE2113_DSAProject.exe
37
a      18
    19
   9
  10
  3
 1
 2
 7
51
24
11
13
27
Press any key to continue . . .

The symbols with their codes are as follows
a      00
g      010
d      01100
b      01101
c      0111
h      100
f      101
e      11
Press any key to continue . . .
Enter the string to be encoded by Huffman Coding: adbgghe
Encoded code for the input string 'adbgghe' is
00011000110101001010011
Press any key to continue . . .
```

CONCLUSION

The original representation has 88 bytes (704 bits) and the new representation have only 26.5 bytes (212 bits) that is 69.88% smaller than the original. So the Huffman Coding turns to be a simple and efficient way to encode data into a short representations without losing any piece of information.

In this age of ever increasing information, the need for storage devices and server for both online and offline usage has seen a great hike. The data compression algorithms, like Huffman Coding, can prove to be very crucial in minimizing the storage demands while keeping the information intact.

This implementation of Huffman Code using an efficient greedy algorithm demonstrates the lossless text compression while getting about 70% compression on a test case. These values may vary for different sets of input.

REFERENCES

- ❖ <http://www.itl.nist.gov/div897/sqg/dads/HTML/codingTree.html>
 - ❖ <http://encyclopedia2.thefreedictionary.com/Huffman+tree>
 - ❖ http://en.wikipedia.org/wiki/Huffman_coding
-