HUFFMAN CODES AND ITS IMPLEMENTATION

Submitted by

KESARWANI AASHITA

Int. M.Sc. in Applied Mathematics (3RD YEAR)

Under the Guidance of

Prof. Sunita Gakkhar

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

ROORKEE-247667 (INDIA)

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the project entitled **"Huffman Coding and its Implementation"** submitted in the **Department of Mathematics** of the **Indian Institute of Technology Roorkee, Roorkee** is an authentic record of my own work carried out for the requirement of the **M.Sc. Integrated (Applied Mathematics)** course **MA-308** in **Autmn Semester 2009-2010**.

The matter embodied in this project has not been submitted by me for the award of any other degree of this or any other institute/university.

Signature
Kesarwani Aashita
M.Sc Integrated Applied Maths (IIIrd year)
I I T Roorkee
EnrollNO: 072070

Date: 28/04/2010

CERTIFICATE

This is certified that the above statement made by the candidate is correct to the best of my knowledge.

Prof Sunita Gakkhar
Course Coordinator
Department of Mathematics,
I I T Roorkee
Roorkee- 247 667 (INDIA)

Date: 28/04/2010
Place: Roorkee

Acknowledgement

I am heartily thankful to my guide, Dr. Sunita Gakkhar, whose encouragement, guidance and support enabled me to develop an understanding of the subject and made it possible for me to complete this project. I offer my regards to all the respected professors that have taught me since their contribution towards my knowledge has been immense enabling me to work on this project.

-Kesarwani Aashita

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.

CONTENTS

1. 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.

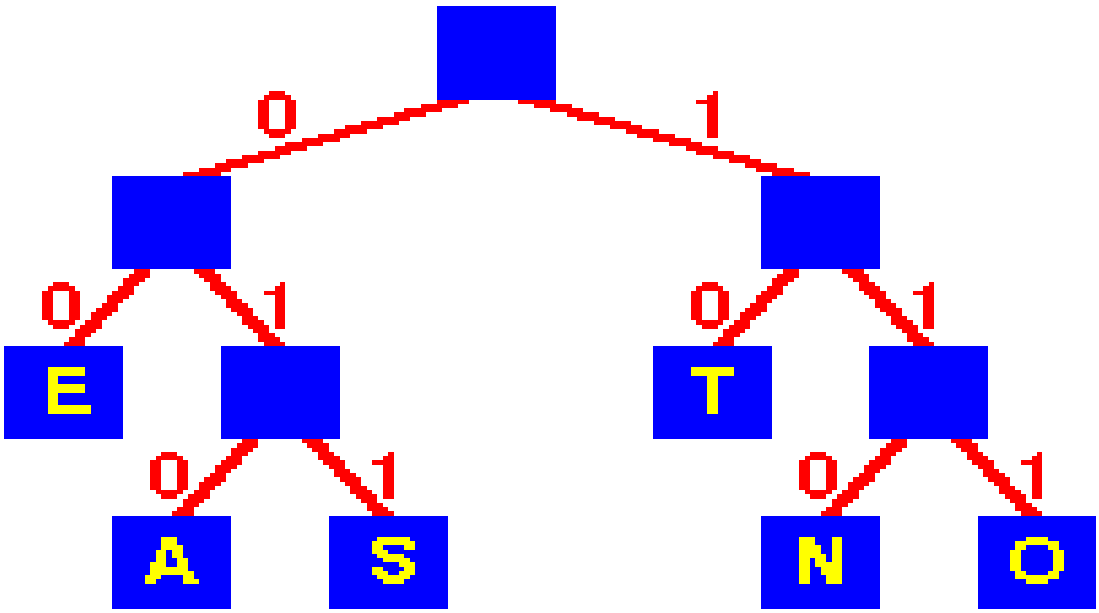
The number of occurrence of a symbol in a string is called its frequency. When there is considerable difference in the frequencies of different symbols in a string, variable length codes can be assigned to the symbols based on their relative frequencies. The most common characters can be represented 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.

2. 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 codeword. 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.

PROBLEM DEFINITION:-

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 codewords) with minimum expected codeword 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):

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

Definition of 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").

3. Implementation of Huffman Coding

a) PROGRAM

```
#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; // Number of symbols
   public:
    minHeap();
    void heapify(int i);
    BinaryTree dequeue(); // Returns the first Binary Tree of the min heap and
                          // then heapify the array of Binary trees in order of the
                          //frequencies of their root nodes.
    void enqueue(BinaryTree b); // To insert another Binary tree
                          // and then heapify the array of Binary trees
    void print();
    friend class HuffmanCode;
};
```

```
class HuffmanCode
{
   private:
      BinaryTree HuffmanTree; // (a minimum weighted external path length tree)
     public:
        HuffmanCode();
};
HuffmanCode::HuffmanCode()
{
   minHeap Heap;
   // Huffman Tree is build from bottom to top.
   // The symbols with lowest frequency are at the bottom of the tree
   // that leads to longer codes for lower frequency symbols and hence
   // shorter codes for higher frequency symbol giving OPTIMAL code length.
   while (Heap.T[0].root->freq>1)
   {
            // The first two trees with min. priority (i.e. frequency) are taken and

            BinaryTree l=Heap.dequeue();
            cout<<"\nAfter dequeueing "<<l.root->freq<<endl;
            Heap.print();
            BinaryTree r=Heap.dequeue();
            cout<<"\nAfter dequeueing "<<r.root->freq<<endl;
            Heap.print();
   // a new tree is constructed taking the above trees as left and right sub-trees
   // with the frequency of root node as the sum of frequencies of left & right child.
            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;
            // then it is inserted in the array and array is heapified again.
            // Deletion and Insertion at an intermediate step is facilitated in heap-sort.
            Heap.enqueue(HuffmanTree);
```

```
            cout<<"\nAfter enqueueing "<<l.root->freq<<"+"<<r.root->freq<<"=
"<<HuffmanTree.root->freq<<endl;
            Heap.print();
    }
    //The process continues till only one tree is left in the array of heap.
    cout<<"\nThe process is completed and Huffman Tree is obtained\n";
    HuffmanTree=Heap.T[1]; // This tree is our HuffmanTree used for coding
    delete []Heap.T;
    cout<<"Traversal of Huffman Tree\n\n";
    HuffmanTree.print();
    cout<<"\nThe symbols with their codes are as follows\n";
    HuffmanTree.assign_code(0); // Codes are assigned to the symbols
    cout<<"Enter the string to be encoded by Huffman Coding: ";
    char *str;
    str=new char[30];
    cin>>str;
    HuffmanTree.encode(str);
    cout<<"Enter the code to be decoded by Huffman Coding: ";
    char *cd;
    cd=new char[50];
   cin>>cd;
    int length;
    cout<<"Enter its code length: ";
    cin>>length;
    HuffmanTree.decode(cd,length);
    delete [ ]cd;
    delete [ ]str;
}

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 at any time 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 occurence in the string:- ";
     cin>>T[i].root->freq;
     T[i].root->code=NULL;
     T[i].root->Llink=NULL;
                    T[i].root->Rlink=NULL;
            // Initially, all the nodes are leaf nodes and stored as an array of trees.
  }
  cout<<endl;

int i=(int)(n / 2);// Heapification will be started from the PARENT element of
    //the last ( 'n th' ) element in the heap.
    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::dequeue()
{
    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::enqueue(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);
```

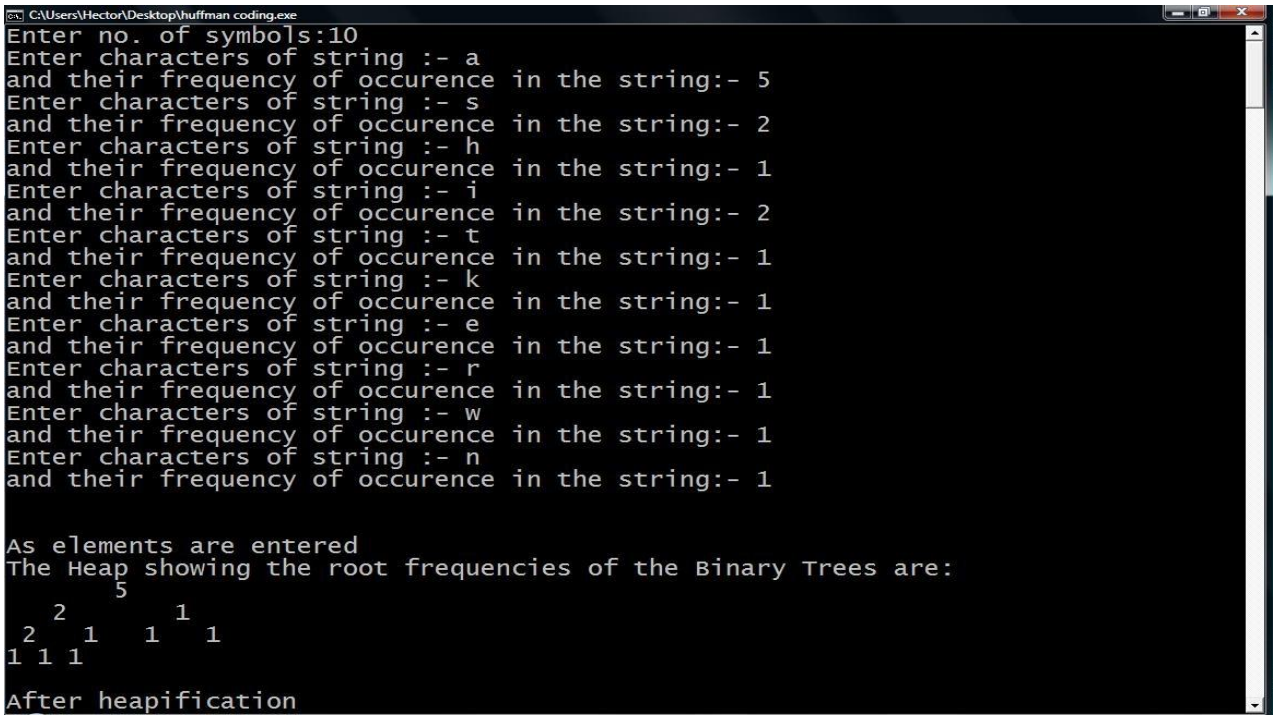
```
    r.root=root->Rlink;
    if (f!=1)
    r.print_code(c);
}

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++;
```

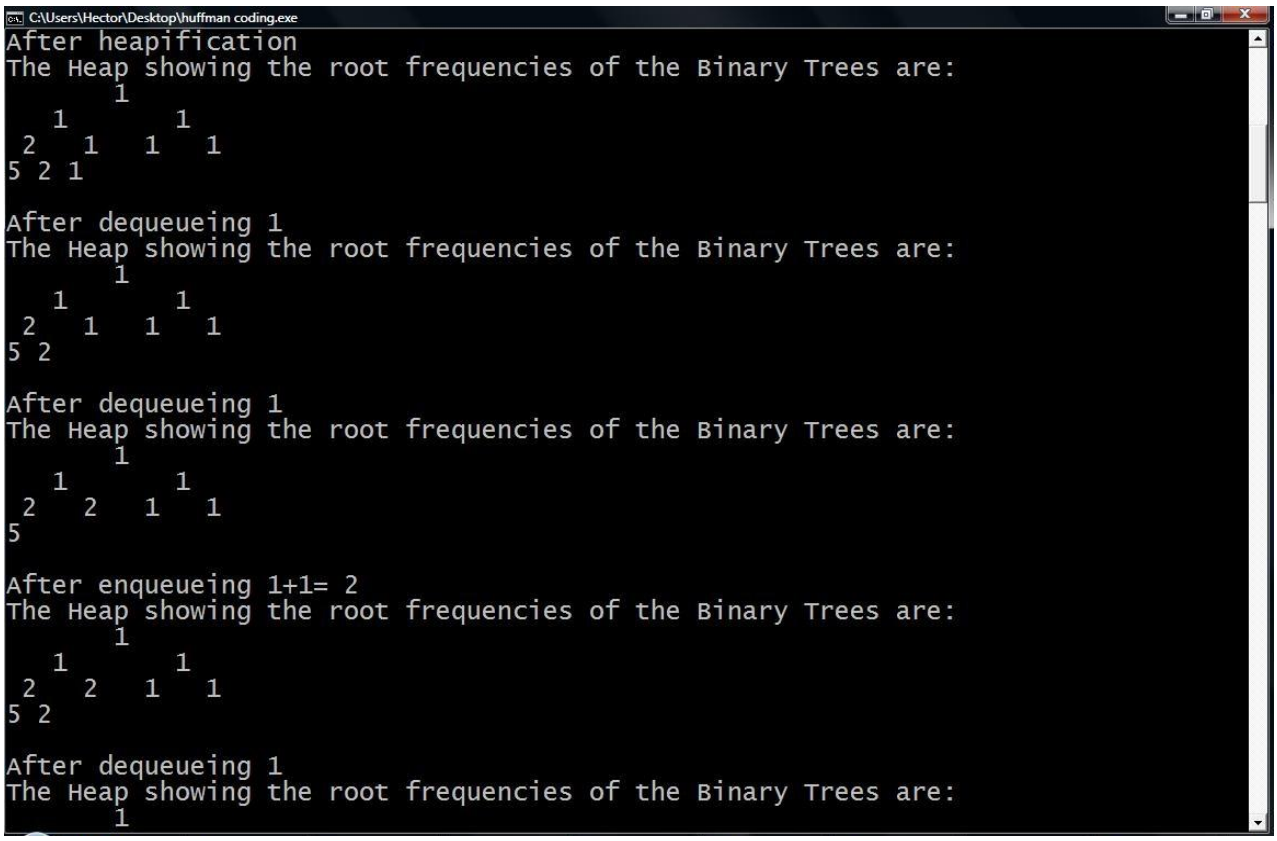
```
                print_symbol(s,f,length);
            }
            i=i+length;
            length=0;
    }
    cout<<endl;
}

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;
```

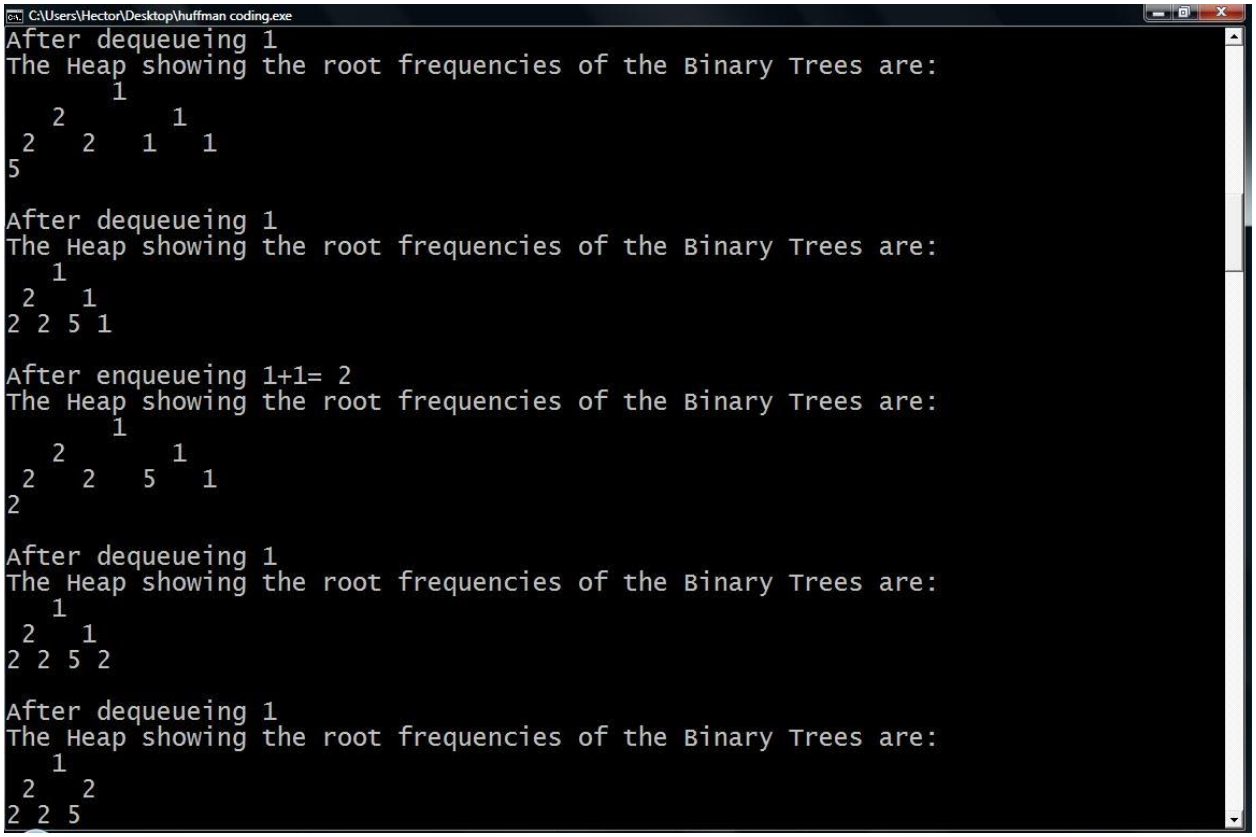
```
    l.root=root->Llink;
    r.root=root->Rlink;
    l.print();
    r.print();
}

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;
}
```

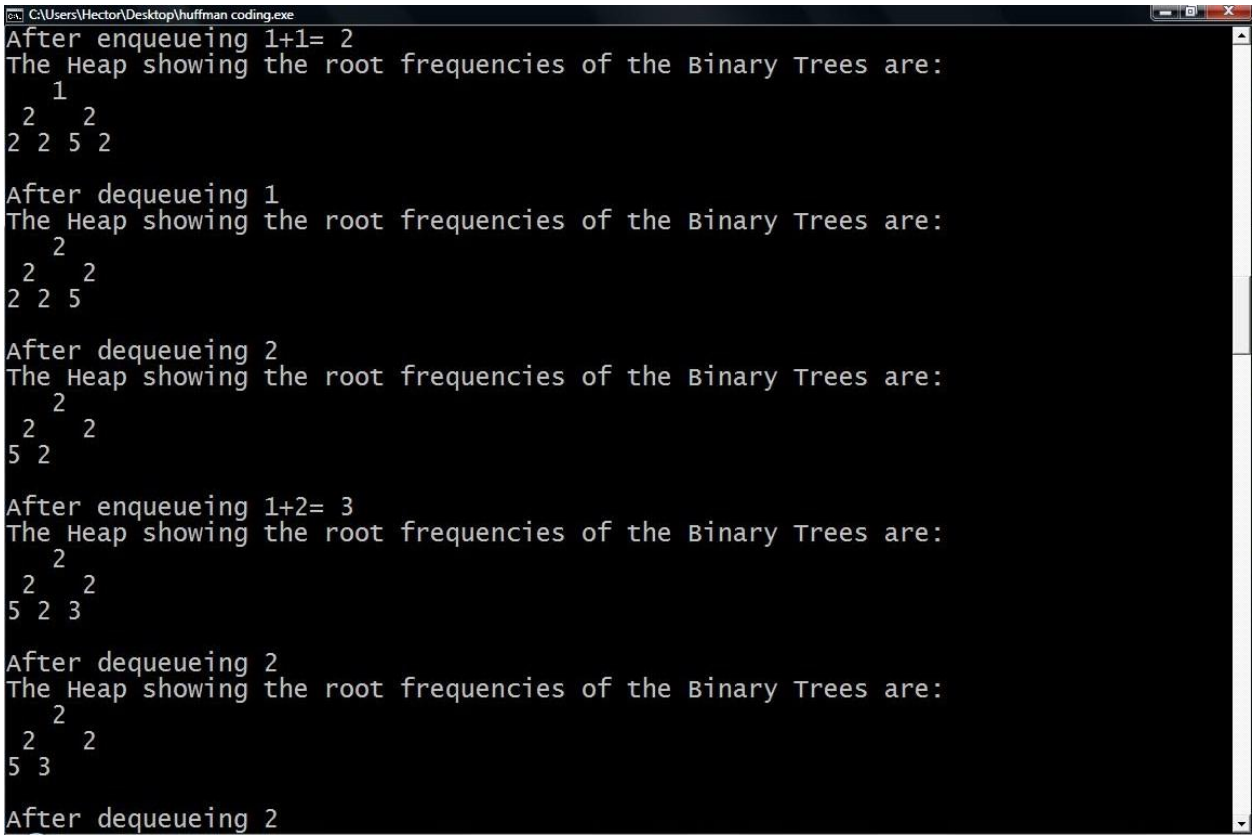
```
void minHeap::print()
{
   cout<<"The Heap showing the root frequencies of the Binary Trees are:\n";
   if (T[0].root->freq==0)
   {
            cout<<endl;
            return;
   }
   int level=1;
   while( T[0].root->freq >= power(2,level) ) // 2^n-1 is the max. no. of nodes
   ///in a complete tree of n levels
   level++;
   if(level==1)
   {
            cout<<T[1].root->freq<<"\n";
            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;
}
int main()
{
   HuffmanCode c;
   system ("pause");
   return 0;}
```

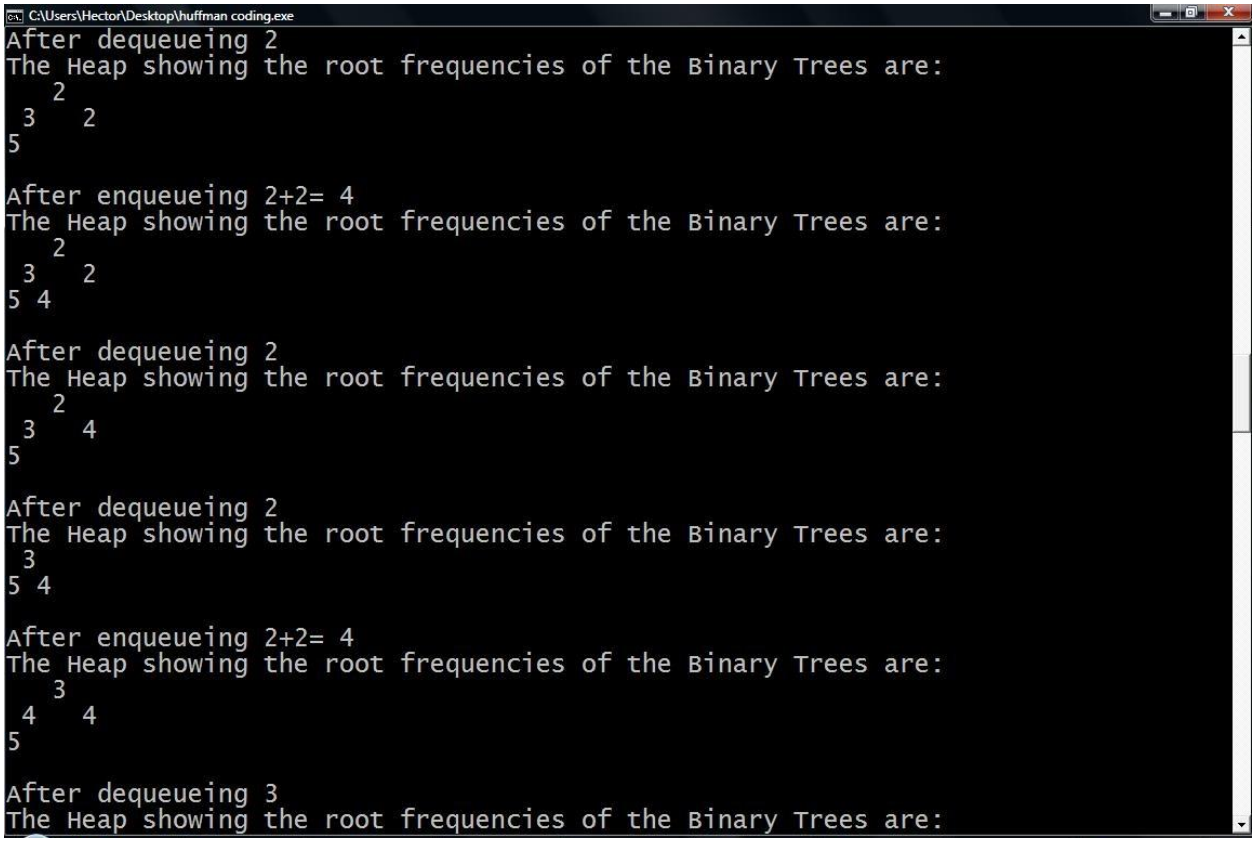
Output

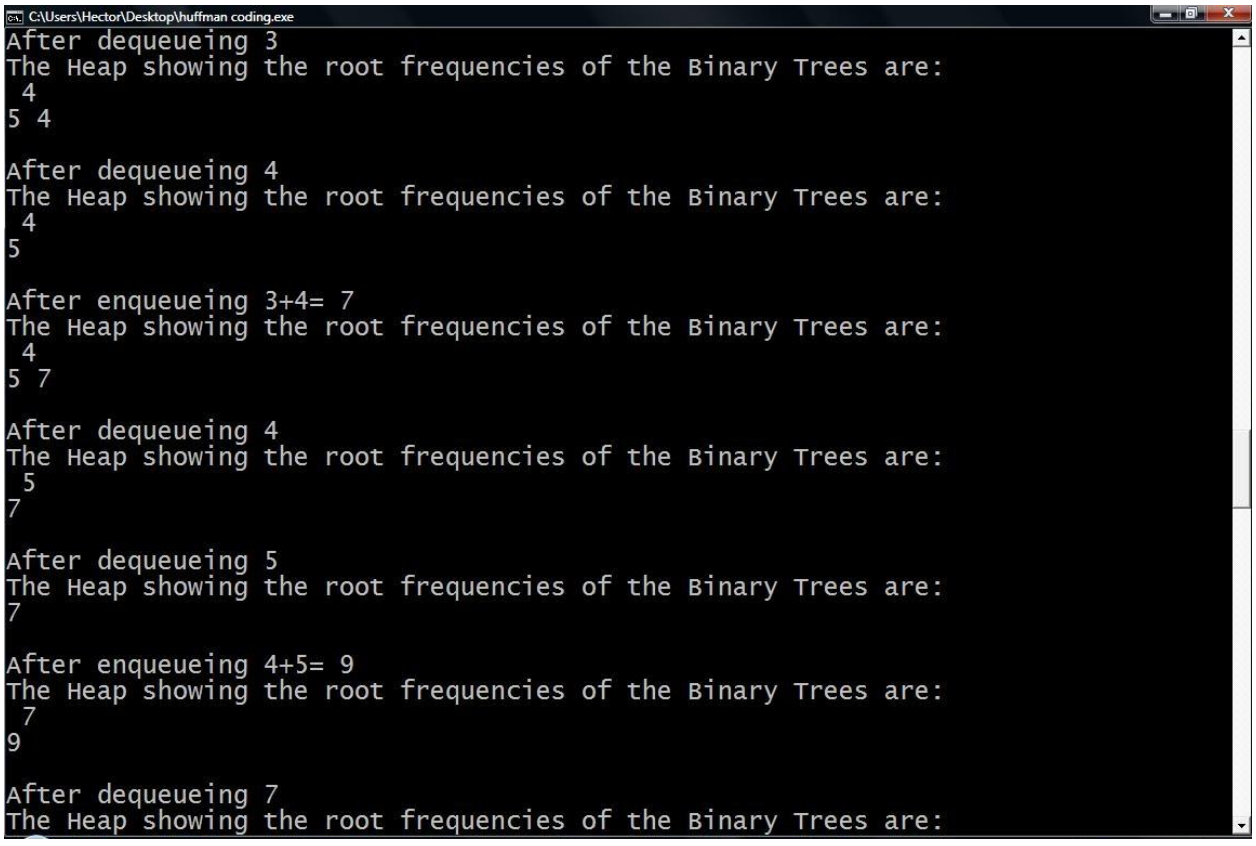

C:\Users\Hector\Desktop\huffman coding.exe
Enter no. of symbols:10
Enter characters of string :- a
and their frequency of occurence in the string:- 5
Enter characters of string :- s
and their frequency of occurence in the string:- 2
Enter characters of string :- h
and their frequency of occurence in the string:- 1
Enter characters of string :- i
and their frequency of occurence in the string:- 2
Enter characters of string :- t
and their frequency of occurence in the string:- 1
Enter characters of string :- k
and their frequency of occurence in the string:- 1
Enter characters of string :- e
and their frequency of occurence in the string:- 1
Enter characters of string :- r
and their frequency of occurence in the string:- 1
Enter characters of string :- w
and their frequency of occurence in the string:- 1
Enter characters of string :- n
and their frequency of occurence in the string:- 1
As elements are entered
The Heap showing the root frequencies of the Binary Trees are:
5
2 1
2 1 1 1
1 1 1
After heapification

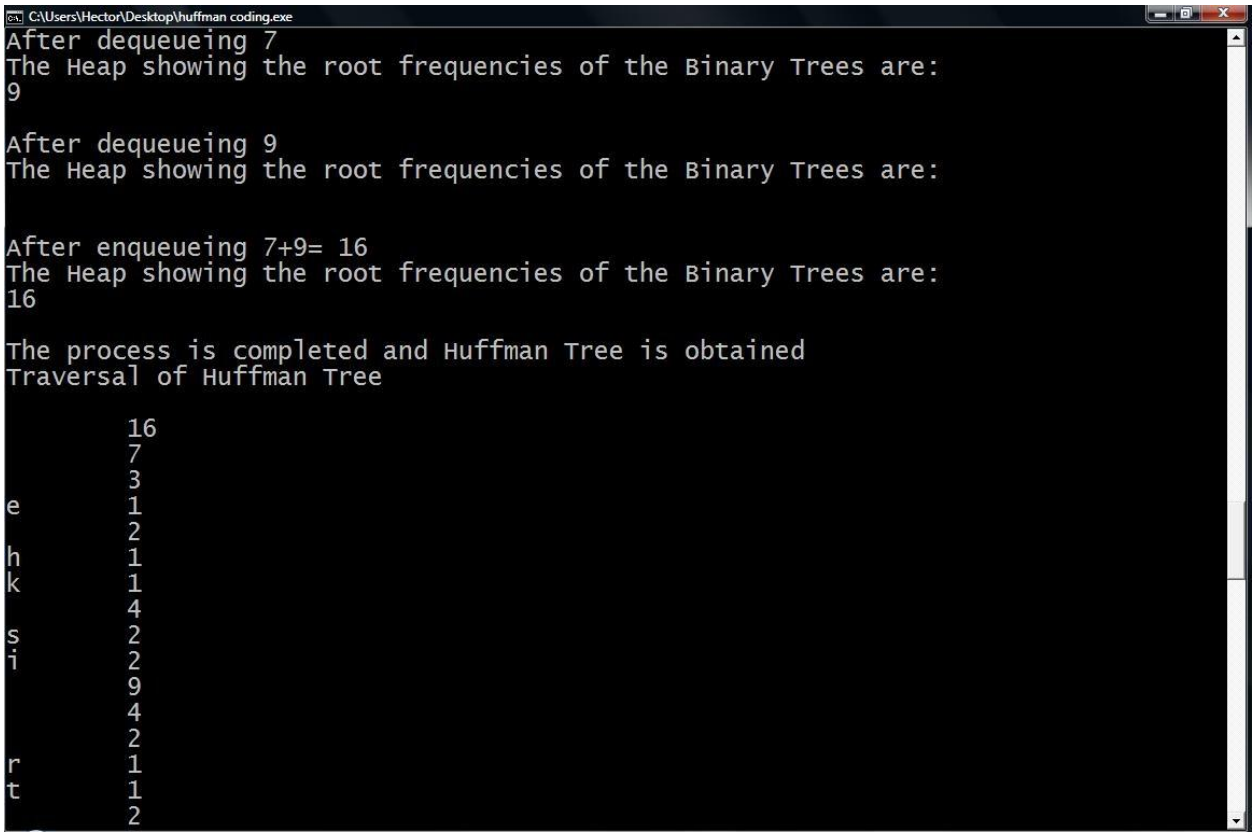

C:\Users\Hector\Desktop\huffman coding.exe
After heapification
The Heap showing the root frequencies of the Binary Trees are:
1
1 1
2 1 1 1
5 2 1
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1
1 1
2 1 1 1
5 2
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1
1 1
2 2 1 1
5
After enqueueing 1+1= 2
The Heap showing the root frequencies of the Binary Trees are:
1
1 1
2 2 1 1
5 2
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1

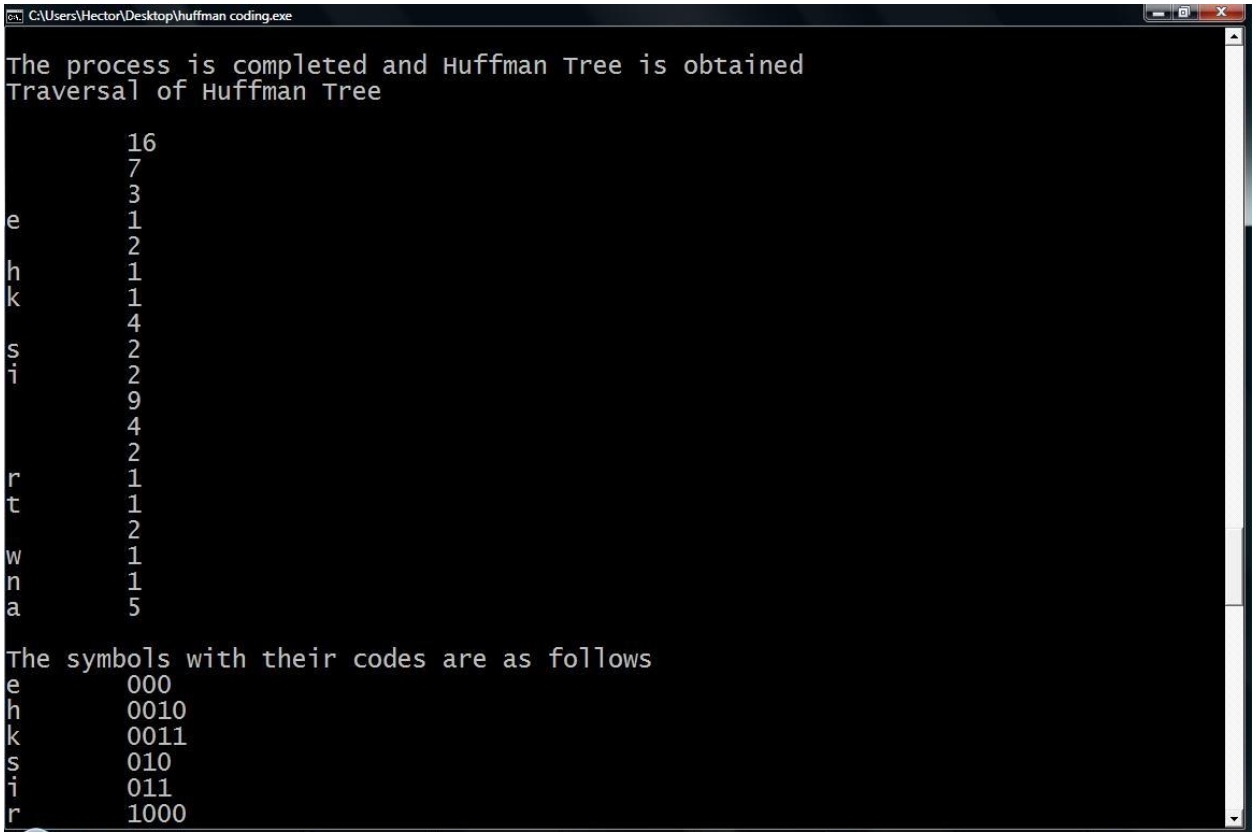

C:\Users\Hector\Desktop\huffman coding.exe
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1
2 1
2 2 1 1
5
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1
2 1
2 2 5 1
After enqueueing 1+1= 2
The Heap showing the root frequencies of the Binary Trees are:
1
2 1
2 2 5 1
2
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1
2 1
2 2 5 2
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
1
2 2
2 2 5

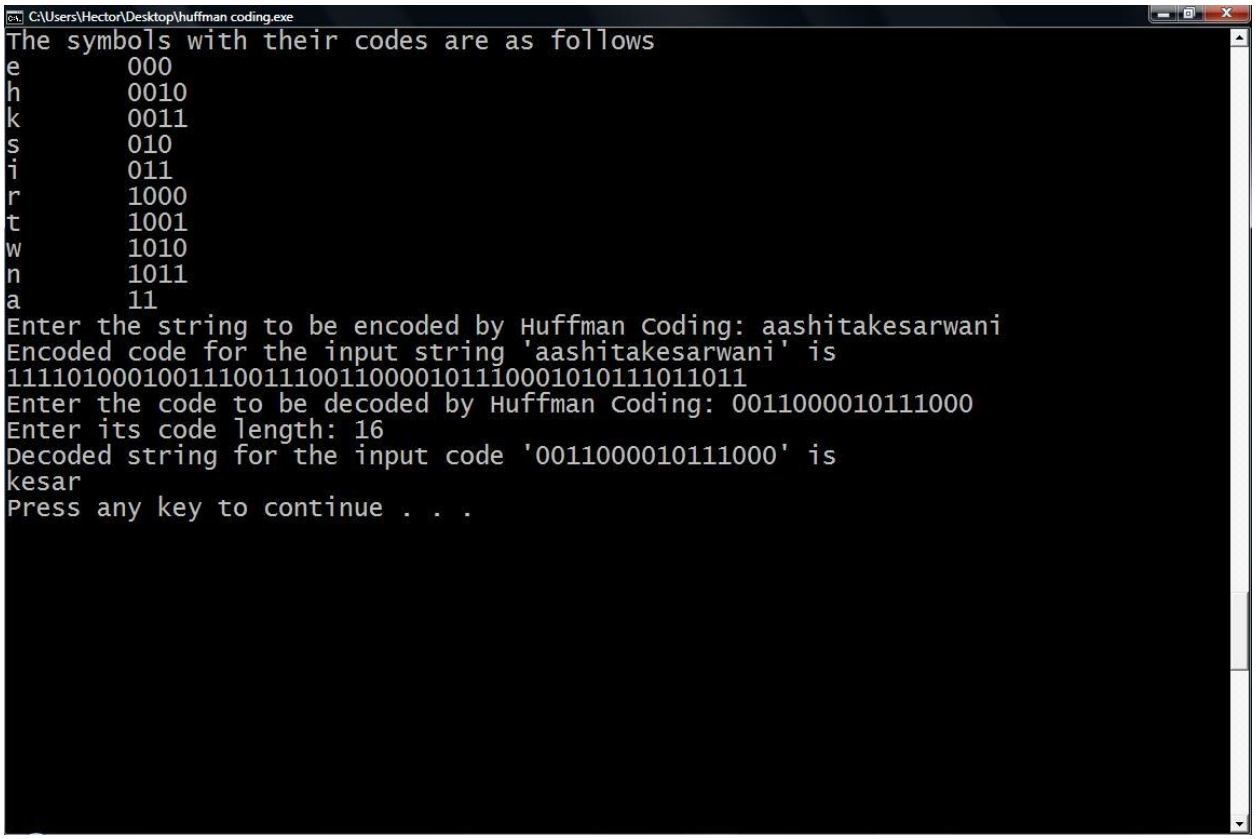

C:\Users\Hector\Desktop\huffman coding.exe
After enqueueing 1+1= 2
The Heap showing the root frequencies of the Binary Trees are:
1
2 2
2 2 5 2
After dequeueing 1
The Heap showing the root frequencies of the Binary Trees are:
2
2 2
2 2 5
After dequeueing 2
The Heap showing the root frequencies of the Binary Trees are:
2
2 2
5 2
After enqueueing 1+2= 3
The Heap showing the root frequencies of the Binary Trees are:
2
2 2
5 2 3
After dequeueing 2
The Heap showing the root frequencies of the Binary Trees are:
2
2 2
5 3
After dequeueing 2


C:\Users\Hector\Desktop\huffman coding.exe
After dequeueing 2
The Heap showing the root frequencies of the Binary Trees are:
2
3 2
5
After enqueueing 2+2= 4
The Heap showing the root frequencies of the Binary Trees are:
2
3 2
5 4
After dequeueing 2
The Heap showing the root frequencies of the Binary Trees are:
2
3 4
5
After dequeueing 2
The Heap showing the root frequencies of the Binary Trees are:
3
5 4
After enqueueing 2+2= 4
The Heap showing the root frequencies of the Binary Trees are:
3
4 4
5
After dequeueing 3
The Heap showing the root frequencies of the Binary Trees are:


C:\Users\Hector\Desktop\huffman coding.exe
After dequeueing 3
The Heap showing the root frequencies of the Binary Trees are:
4
5 4
After dequeueing 4
The Heap showing the root frequencies of the Binary Trees are:
4
5
After enqueueing 3+4= 7
The Heap showing the root frequencies of the Binary Trees are:
4
5 7
After dequeueing 4
The Heap showing the root frequencies of the Binary Trees are:
5
7
After dequeueing 5
The Heap showing the root frequencies of the Binary Trees are:
7
After enqueueing 4+5= 9
The Heap showing the root frequencies of the Binary Trees are:
7
9
After dequeueing 7
The Heap showing the root frequencies of the Binary Trees are:


C:\Users\Hector\Desktop\huffman coding.exe
After dequeueing 7
The Heap showing the root frequencies of the Binary Trees are:
9
After dequeueing 9
The Heap showing the root frequencies of the Binary Trees are:
After enqueueing 7+9= 16
The Heap showing the root frequencies of the Binary Trees are:
16
The process is completed and Huffman Tree is obtained
Traversal of Huffman Tree
16
7
3
e 1
2
h 1
k 1
4
s 2
i 2
9
4
2
r 1
t 1
2


C:\Users\Hector\Desktop\huffman coding.exe
The process is completed and Huffman Tree is obtained
Traversal of Huffman Tree
16
7
3
e 1
2
h 1
k 1
4
s 2
i 2
9
4
2
r 1
t 1
2
w 1
n 1
a 5
The symbols with their codes are as follows
e 000
h 0010
k 0011
s 010
i 011
r 1000


C:\Users\Hector\Desktop\huffman coding.exe
The symbols with their codes are as follows
e 000
h 0010
k 0011
s 010
i 011
r 1000
t 1001
w 1010
n 1011
a 11
Enter the string to be encoded by Huffman Coding: aashitakesarwani
Encoded code for the input string 'aashitakesarwani' is
11110100010011100111001100001011100010101111011011
Enter the code to be decoded by Huffman Coding: 0011000010111000
Enter its code length: 16
Decoded string for the input code '0011000010111000' is
kesar
Press any key to continue . . .

4. Variations of Huffman Coding:

a) n-ary Huffman coding

The **n-ary Huffman** algorithm uses the {0, 1, ... , n − 1} alphabet to encode message and build an n-ary tree.

b) Adaptive Huffman coding

It calculates the probabilities dynamically based on recent actual frequencies in the source string. This is somewhat related to the LZ family of algorithms.

c) Huffman template algorithm

The **Huffman template algorithm** enables one to use any kind of weights (costs, frequencies, pairs of weights, non-numerical weights) and one of many combining methods (not just addition).

d) Optimal alphabetic binary trees (Hu-Tucker coding)

In the alphabetic version, the alphabetic order of inputs and outputs must be identical.This is also known as the **Hu-Tucker** problem, after the authors of the paper presenting the first linearithmic solution to this optimal binary alphabetic problem, which has some similarities to Huffman algorithm, but is not a variation of this algorithm. These optimal alphabetic binary trees are often used as binary search trees.

e) The canonical Huffman code

If weights corresponding to the alphabetically ordered inputs are in numerical order, the Huffman code has the same lengths as the optimal alphabetic code, which can be found from calculating these lengths, rendering Hu-Tucker coding unnecessary. The code resulting from numerically (re-)ordered input is sometimes called the canonical Huffman code and is often the code used in practice, due to ease of encoding/decoding. The technique for finding this code is sometimes called **Huffman-Shannon-Fano coding**, since it is optimal like Huffman coding, but alphabetic in weight probability, like Shannon-Fano coding.

5. Applications:

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 nonbinary 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.

6. References

- Sartaj Sahani: Data structures, Algorithms and Applications in C++
- http://www.itl.nist.gov/div897/sqg/dads/HTML/codingTree.html
- http://encyclopedia2.thefreedictionary.com/Huffman+tree
- http://en.wikipedia.org/wiki/Huffman_coding