

**A Project Report On**

**‘Modified Huffman Coding used for Encryption and Decryption’**

**Subject** – Data Structures and Algorithm

**Course Code** – CSE

**Team Members :**

Amreeta Koner- 17BCE2033

Under the supervision of Prof. Anbarasi M.

**ABSTRACT**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are [Prefix Codes](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

# INTRODUCTION

## Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps #2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

# PURPOSE OF HUFFMAN CODING

Huffman coding is based on the frequency of occurrence of a data item (pixel in images). The principle is to use a lower number of bits to encode the data that occurs more frequently. Codes are stored in a ***Code Book*** which may be constructed for each image or a set of images. In all cases the code book plus encoded data must be transmitted to enable decoding.

# THE BASIC ALGORITHM

* + Huffman coding is a form of statistical coding
  + Not all characters occur with the same frequency!
  + Yet all characters are allocated the same amount of space
  + 1 char = 1 byte, be it e or x or ‘space’
  + Any savings in tailoring codes to frequency of character?
  + Code word lengths are no longer fixed like ASCII.
  + Code word lengths vary and will be shorter for the more frequently used characters.

# THE MODIFIED ALGORITHM

* + Scan text to be compressed and tally occurrence of all characters.
  + Sort or prioritize characters based on number of occurrences in text.
  + Build Huffman code tree based on prioritized list.
  + Perform a traversal of tree to determine all code words.
  + Scan text again and create new file using the Huffman codes.

# BUILDING A TREE

*Scan the original text*

* Consider the following short text:

*Eerie eyes seen near lake.*

* Count up the occurrences of all characters in the text
* What characters are present?

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **E** | **e** | **r** | **i** | **space** | | |
| **y** | **s** | **n** | **a** | **r** | **l** | **k** |

* + What is the frequency of each character in the text?

|  |  |  |  |
| --- | --- | --- | --- |
| **Char** | **Freq.** | **Char** | **Freq.** |
| **E** | 1 | **s** | 2 |
| **e** | 8 | **n** | 2 |
| **r** | 2 | **a** | 2 |
| **i** | 1 | **l** | 1 |
| **space** | 4 | **k** | 1 |
| **y** | 1 | **.** | 1 |

# BUILDING A TREE

*Prioritize characters*

* + Create binary tree nodes with character and frequency of each character
  + Place nodes in a priority queue-The lower the occurrence, the higher the priority in the queue
  + *Uses binary tree nodes*

public class HuffNode

{

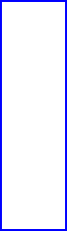
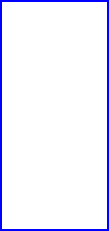
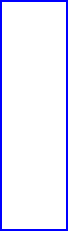
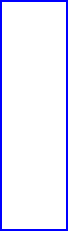
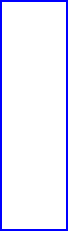
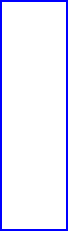
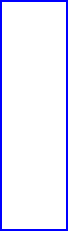
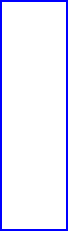
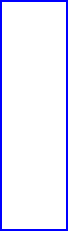
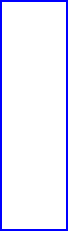
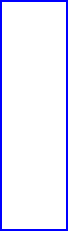
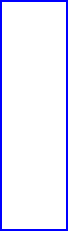
public char myChar; public int myFrequency;

public HuffNode myLeft, myRight;

}

priorityQueue myQueue;

* + The queue after inserting all nodes



**E**

**1**

**i**

**1**

**y**

**1**

**l**

**1**

**k**

**1**

**.**

**1**

**r**

**2**

**s**

**2**

**n**

**2**

**a**

**2**

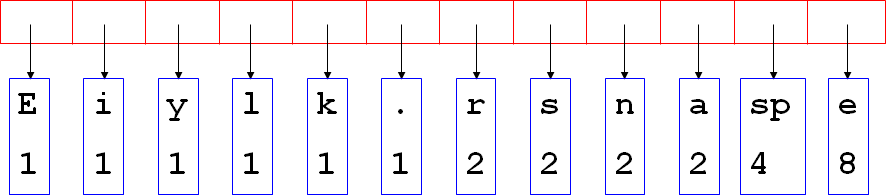
**s p**

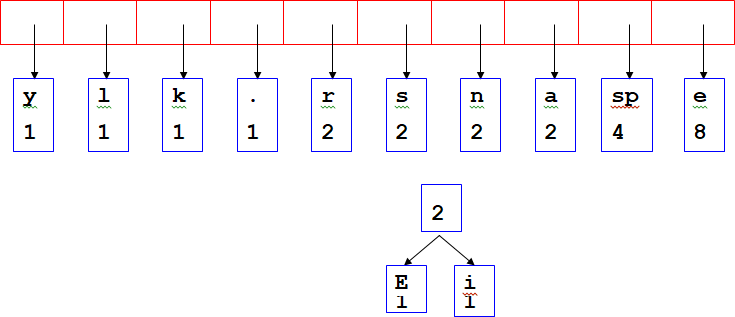
**e**

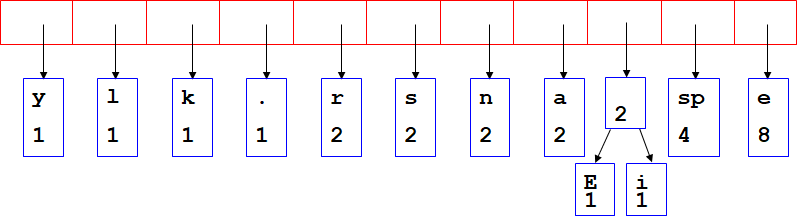
**8**

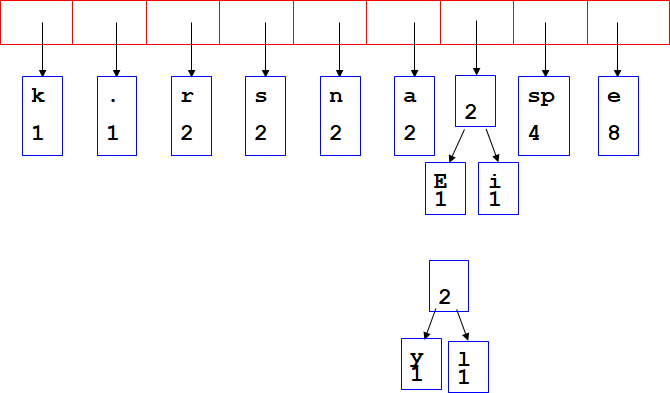
\*Null Pointers are not shown.

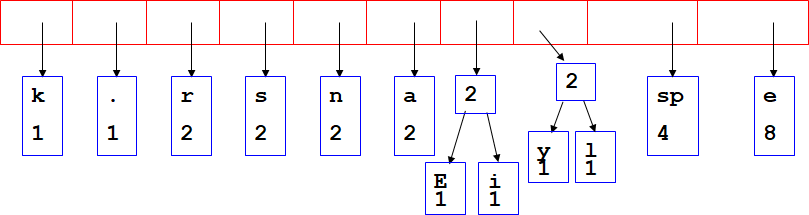
* + *While priority queue contains two or more nodes*
    - Create new node
    - Dequeue node and make it left subtree
    - Dequeue next node and make it right subtree
    - Frequency of new node equals sum of frequency of left and right children
    - Enqueue new node back into queue

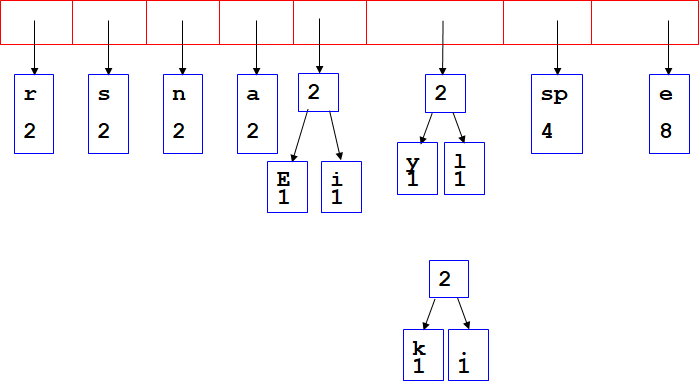


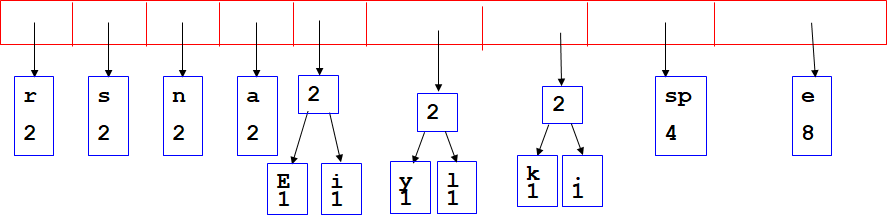


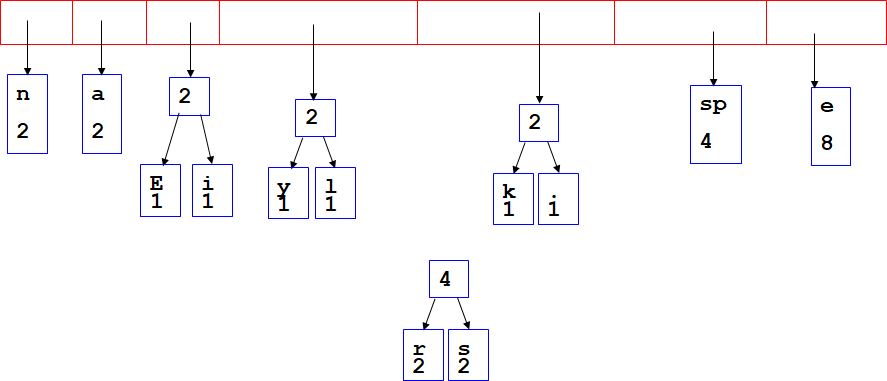


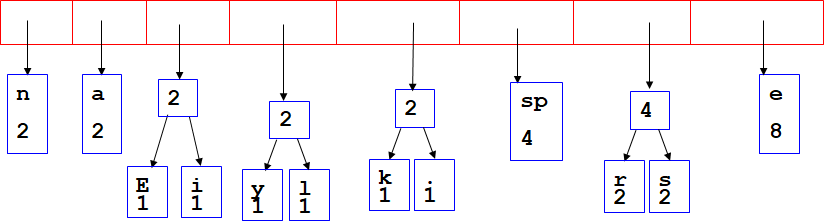


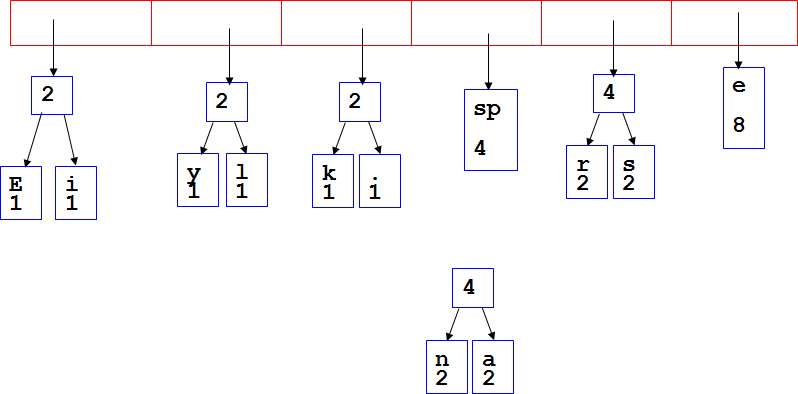


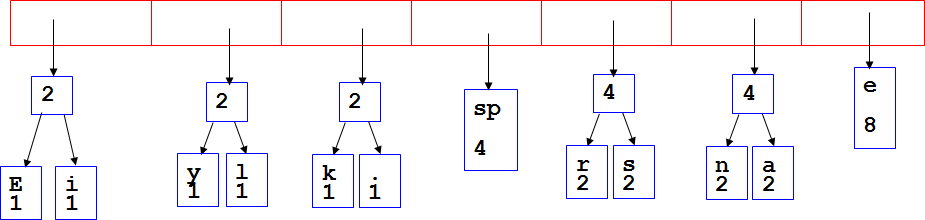


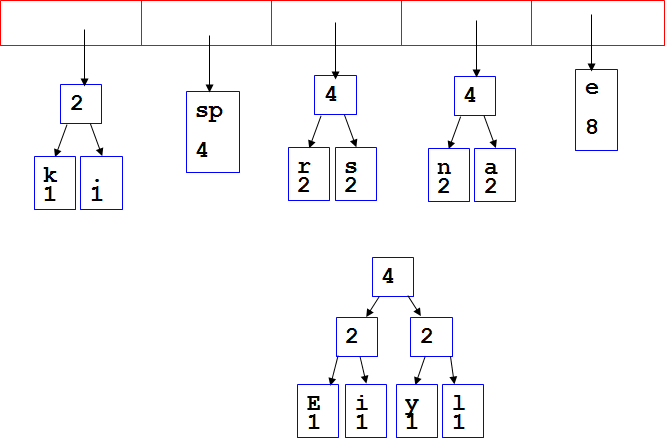


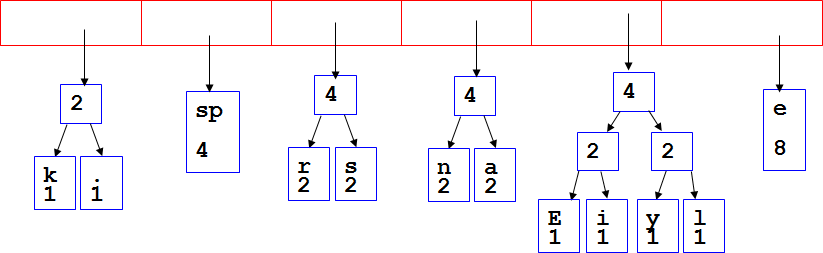


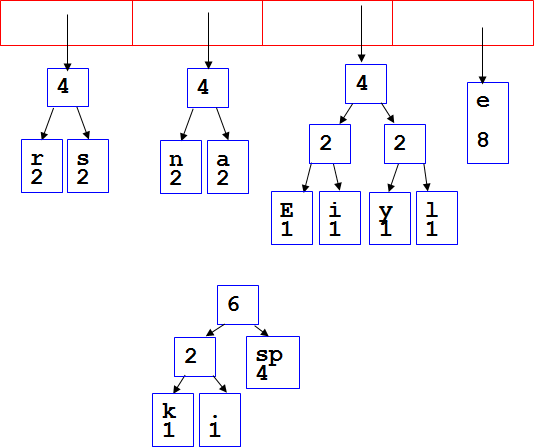


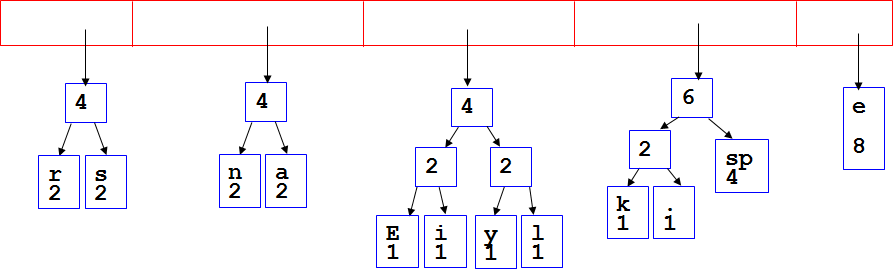


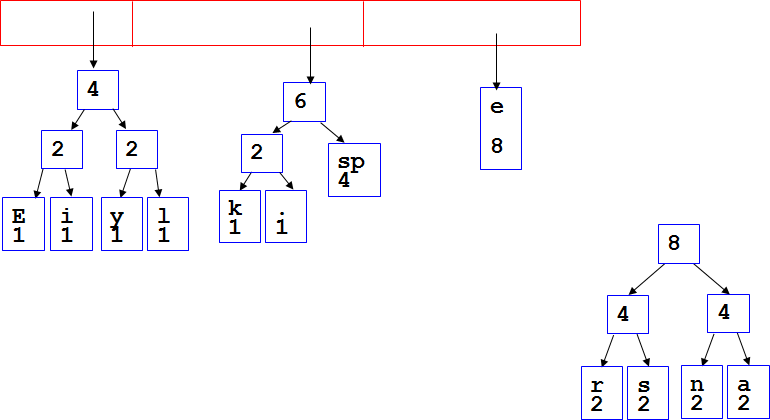


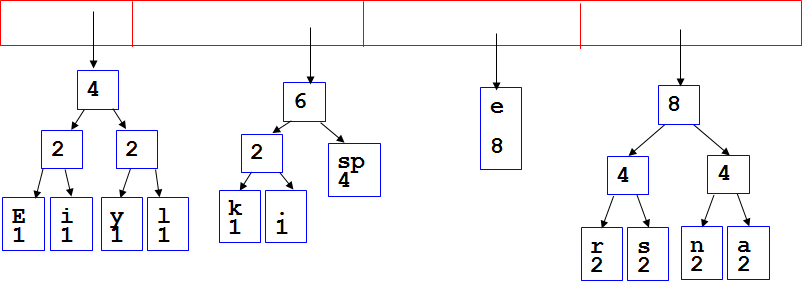


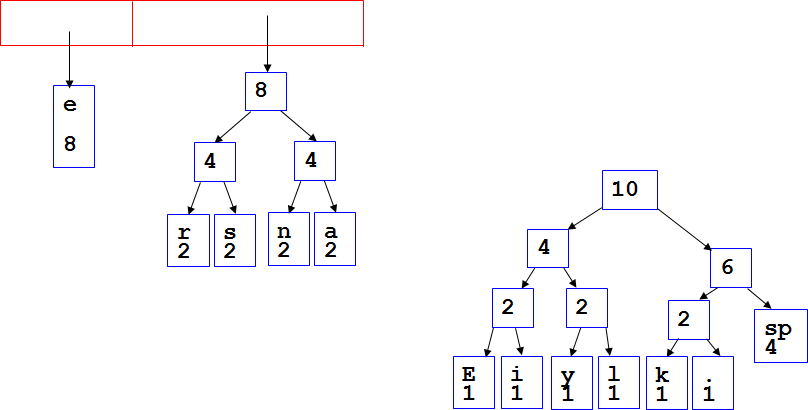


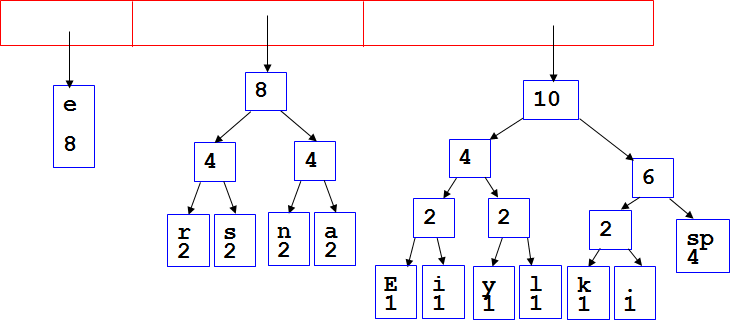


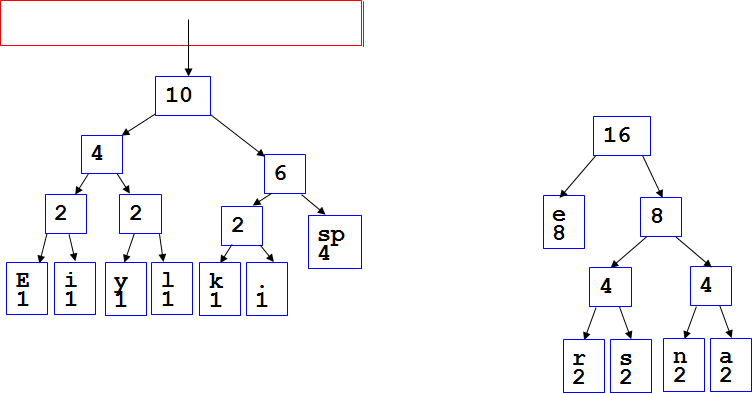


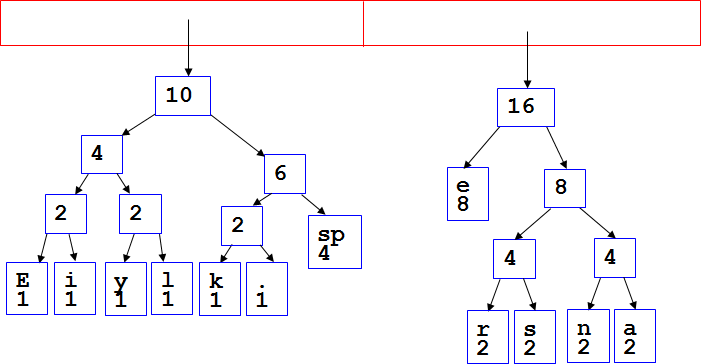


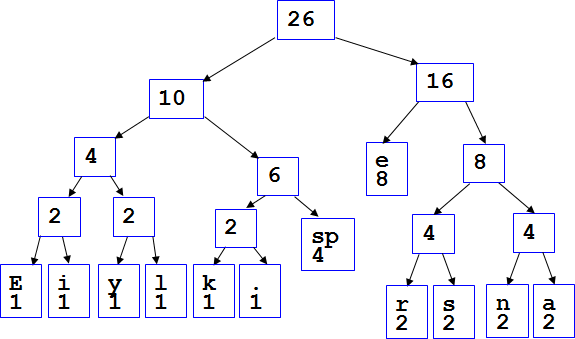


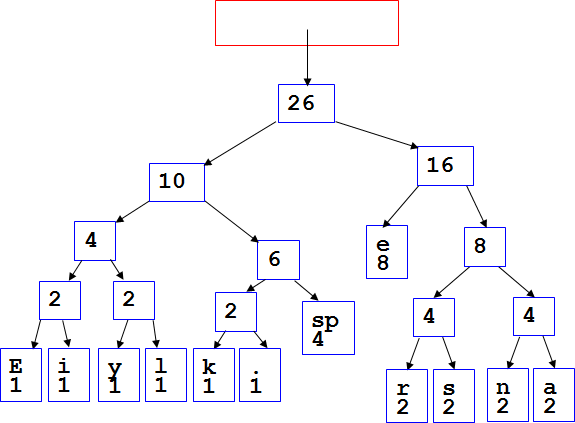






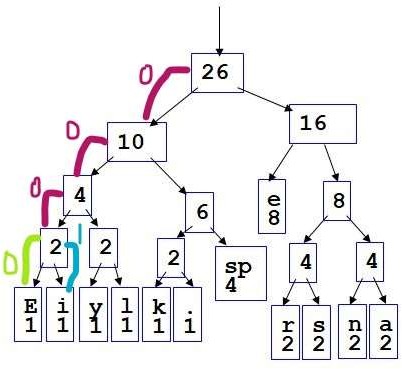






*After enqueueing this node there is only one node left in the priority queue.*

* + Dequeue the single node left in the queue.
  + This tree contains the new code words for each character.
  + Frequency of root node should equal number of characters in text.
  + Eerie eyes seen near lake → 26 characters.
  + Perform a traversal of the tree to obtain new code words
  + Going left is a 0 going right is a 1
  + code word is only completed when a leaf node is reached



|  |  |
| --- | --- |
| **Character** | **Code** |
| **E** | 0000 |
| **i** | 0001 |
| **y** | 0010 |
| **l** | 0011 |
| **k** | 0100 |
| **.** | 0101 |
| **space** | 011 |
| **e** | 10 |
| **r** | 1100 |
| **s** | 1101 |
| **n** | 1110 |
| **a** | 1111 |

* + *Rescan text and encode file using new code words*

000010110000011001110001010110110100

1111101011111100011001111110100100101

* + *Have we made things any better?*
* 73 bits to encode the text
* ASCII would take → 8 \* 26 = 208 bits
* If modified code used 4 bits per character are needed.

- Total bits → 4 \* 26 = 104.

* + *How does receiver know what the codes are?*
* Tree constructed for each text file.

» Considers frequency for each file

» Big hit on compression, especially for smaller files

* Tree predetermined

» based on statistical analysis of text files or file types

» Data transmission is bit based versus byte based

**Encryption and Decryption:**

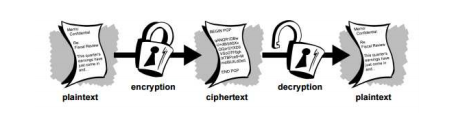
Data that can be read and understood by anyone without any special knowledge about it is called plaintext or clear text.

The method of disguising the plaintext in such a way as to hide the information is called encryption. Encrypting plaintext

results in unreadable gibberish called cipher text. You use encryption to ensure that information is hidden from anyone for

whom it is not intended, even those who can see the encrypted data. The process of reverting cipher text to its original

plaintext is called decryption.



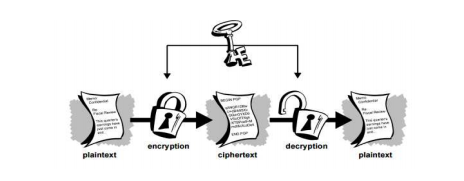
**Conventional Cryptography**

In conventional cryptography, also called secret-key or symmetric-key encryption, one key is used to both encrypt and

decrypt the data. The famous Caesar’s Cipher is an example of this technique. Only the person who knew the “shift by 3” rule

could understand the message. While sending the encrypted data, the key is shared through another secure channel so as

to make it possible for the receiver to decrypt the cypher text into original plaintext. Which makes it a bit insecure.

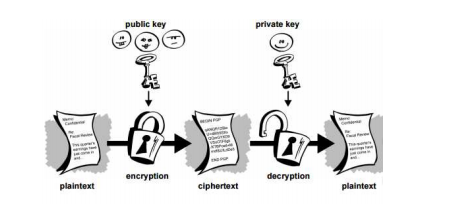


**Public Key Cryptography**

In this technique, there are two keys involved, one for encryption and other for decryption. The receiver already has a

private key which is never used in any communications. Only the public key is sent along with the cipher text. No one without

having the private key can decrypt the code, thus making the communication safer.



**THE PROPOSED APPROACH**

**Encryption Algorithm**

Step 1. Compress the given text using Huffman coding algorithm as explained above.

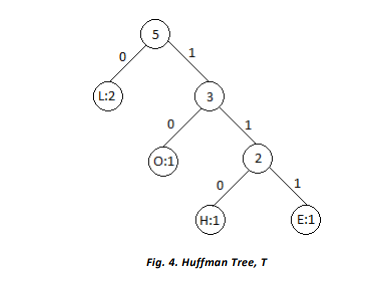
Here is an example to do that:

Let’s say the data being sent is ‘HELLO’.

1. Calculating the frequencies of all the symbols:

H: 1; E: 1; L: 2; O: 1.

2. Constructing the tree:



Code:

H: 110

E: 111

L: 0

O: 10

Therefore, the compressed code becomes, say H = 1101110010.

Step 2. Take two keys ‘A’ and ‘B’. ‘A’ is used as public key, not known to the receiver beforehand and ‘B’ (should be small) is

used as the private key previously known by the receiver.

For our example, let’s say A=1001 and B= 1000.

Step 3. Take the first ‘n’ bits of the compressed code where ‘n’ is the number of bits in ‘A’.

Perform XOR operation on these bits with A and shift A by one bit. Repeat this till you reach the end of the code. The new

code formed is E.

For our example, the operation will be as follows:

1101110010

^1001

0100110010

^1001

0000010010

^1001

0010000010

^1001

0011001010

^1001

0011101110

^1001

0011111100

^1001

0011110101

The encrypted code, E now becomes 0011110101.

Step 4. Multiply ‘E’ by ‘B’ to change E to E\*B.

For our example:

0011110101

\*1000

0011110101000

Thus the final encrypted data to be sent, E= 0011110101000.

While sending the data ‘E’, ‘A’ and the Huffman tree ‘T’ is sent to the receiver so as to enable him to decrypt the message. B

is known to the receiver previously.

**Decryption Algorithm**

Receiver will now have E, A, the Huffman tree T sent by the sender along with B.

Step 1. Divide E by B.

For our example:

E’ = 0011110101000 / 1000 = 0011110101.

Step 2. Starting from the last ‘n’ bits of the obtained code from last step, Apply XOR operation on E’ and A; where ‘n’ is the

number of bits in A. And keep repeating till you reach the first bit of E

0010000010

^1001

0000010010

^1001

0100110010

^1001

1101110010

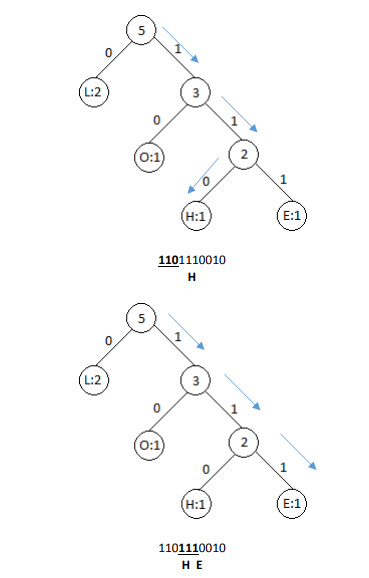
Thus, E” = 1101110010.

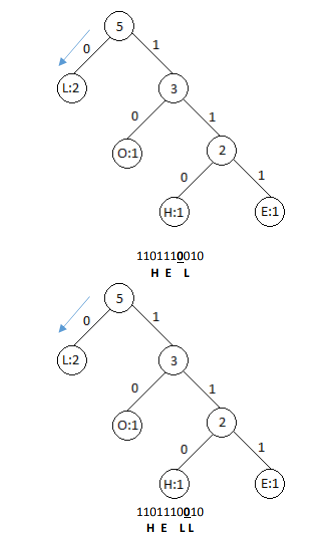
Step 3. Now uncompress the data obtained, for which trace the Huffman tree T, according to the bits of E’. As you reach a

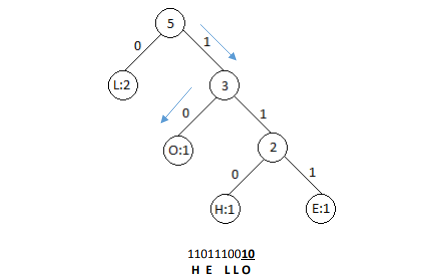
leaf node, note the symbol associated with it and restart from the root node again.

For our example:

Tracing from root node;







Thus, we decrypted the encrypted message.

## CODE:

// C++ program to encode and decode a string using

// Huffman Coding.

#include <bits/stdc++.h>

#define MAX\_TREE\_HT 256

using namespace std;

// to map each character its huffman value

map<char, string> codes;

// to store the frequency of character of the input data

map<char, int> freq;

// A Huffman tree node

struct MinHeapNode

{

char data; // One of the input characters

int freq; // Frequency of the character

MinHeapNode \*left, \*right; // Left and right child

MinHeapNode(char data, int freq)

{

left = right = NULL;

this->data = data;

this->freq = freq;

}

};

// utility function for the priority queue

struct compare

{

bool operator()(MinHeapNode\* l, MinHeapNode\* r)

{

return (l->freq > r->freq);

}

};

// utility function to print characters along with

// there huffman value

void printCodes(struct MinHeapNode\* root, string str)

{

if (!root)

return;

if (root->data != '$')

cout << root->data << ": " << str << "\n";

printCodes(root->left, str + "0");

printCodes(root->right, str + "1");

}

// utility function to store characters along with

// there huffman value in a hash table, here we

// have C++ STL map

void storeCodes(struct MinHeapNode\* root, string str)

{

if (root==NULL)

return;

if (root->data != '$')

codes[root->data]=str;

storeCodes(root->left, str + "0");

storeCodes(root->right, str + "1");

}

// STL priority queue to store heap tree, with respect

// to their heap root node value

priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>, compare> minHeap;

// function to build the Huffman tree and store it

// in minHeap

void HuffmanCodes(int size)

{

struct MinHeapNode \*left, \*right, \*top;

for (map<char, int>::iterator v=freq.begin(); v!=freq.end(); v++)

minHeap.push(new MinHeapNode(v->first, v->second));

while (minHeap.size() != 1)

{

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

top = new MinHeapNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

storeCodes(minHeap.top(), "");

}

// utility function to store map each character with its

// frequency in input string

void calcFreq(string str, int n)

{

for (int i=0; i<str.size(); i++)

freq[str[i]]++;

}

// function iterates through the encoded string s

// if s[i]=='1' then move to node->right

// if s[i]=='0' then move to node->left

// if leaf node append the node->data to our output string

string decode\_file(struct MinHeapNode\* root, string s)

{

string ans = "";

struct MinHeapNode\* curr = root;

for (int i=0;i<s.size();i++)

{

if (s[i] == '0')

curr = curr->left;

else

curr = curr->right;

// reached leaf node

if (curr->left==NULL and curr->right==NULL)

{

ans += curr->data;

curr = root;

}

}

// cout<<ans<<endl;

return ans+'\0';

}

// Driver program to test above functions

int main()

{

string str;

cout << "Enter String";

cin >> str;

string encodedString, decodedString;

calcFreq(str, str.length());

HuffmanCodes(str.length());

cout << "Character With there Frequencies:\n";

for (auto v=codes.begin(); v!=codes.end(); v++)

cout << v->first <<' ' << v->second << endl;

for (auto i: str)

encodedString+=codes[i];

cout << "\nEncoded Huffman data:\n" << encodedString << endl;

decodedString = decode\_file(minHeap.top(), encodedString);

cout << "\nDecoded Huffman Data:\n" << decodedString << endl;

return 0;

}

## 

* + ***Time complexity:***
* O(nlogn) where n is the number of unique characters.
* If there are n nodes, extractMin() is called 2\*(n – 1) times.
* extractMin() takes O(logn) time as it calles minHeapify().
* So, overall complexity is O(nlogn).

## Applications Of Huffman Code:

* Image Compression: Generation of a Huffman code for the set of values that any pixel may take.
* In Window operating system, similar encoding is used to conpress text files
* Text compression in ZIP files
* JPEG fram compression
* MPEG format compression
* FAX transmission
* Electronic Books
* Application of Encryption and Decryption:
* Secure communications. The most obvious use of cryptography, and the one that all of us use frequently, is encrypting communications between us and another system.
* End-to-end Encryption. Email is one area where encryption is not widely in use.
* Storing Data.
* Storing Passwords

## Advantages of Huffman Coding:

» The ultimate purpose of Huffman coding is to give certain character a less bit number, so space is saved. If you assign less number or bits or **shorter code** words for **most frequently** used symbols you will be saving a lot of storage space

» If the frequency table is somehow wrong, the Huffman algorithm will still give you a valid encoding, but the encoded text would be longer than it could have been if you had used a correct frequency table. This is usually not a problem, because we usually create the frequency table based on the actual text that is to be encoded, so the frequency table will be "perfect" for the text that we are going to encode.

» They are prefix codes. The prefix code part is useful because it means that no code is the prefix of another, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

» Another advantage of Huffman algorithm is that you can use it for any alphabet starting from [0, 1] finished Chinese hieroglyphs

» Statistic table can't be wrong, because in general Huffman algorithm, analyze hole text at the beginning, and builds frequent-statistics of the given text

» Huffman algorithm uses the advantage of a given text. As an example, if E is most frequent letter in English in general, that doesn't mean that E is most frequent in a given text for a given author.

* ***Benefits by Encryption and Decryption through Huffman Coding*:**

• It is easy to understand and implement as simple binary operations are being used.

• Huffman compression makes encryption easy by reducing the length of data to be encrypted.

• It can be used by the people who have no preexisting security arrangements for exchanging data.

• Due to the use of a random private key it is computationally infeasible to decrypt the data.

## References:

* Nigam Sangwan, “Text Encryption with Huffman Compression”, International Journal of Computer Applications, vol. 54, no.6, September 2012.
* An Introduction to Cryptography, 1999. [Online]
* Huffman Coding, 2011. [Online] Available: <http://en.wikipedia.org/wiki/Huffman_coding>
* Huffman Coding, 2014. [Online] Available: <http://www.columbia.edu/~cs2035/courses/csor4231.F11/huff.pdf>
* Charles P. Pfleeger, Shari Lawrence Pfleeger, Security in Computing, 3rd Ed. Prentice Hall Professional, 2003.
* Wikipedia <http://en.wikipedia.org/wiki/Huffman_coding>
* Stack exchange [https://softwareengineering.stackexchange.com](https://softwareengineering.stackexchange.com/)
* Book: Introduction to Algorithms-By Thomas H. Cormen