International Research Journal in Global Engineering and Sciences. (IRJGES)
Vol. 1, No. 1, Mar-May, 2016 | ISSN : 2456-172X

Reliable Implementation of Encoding and Decoding based on Huffman Code using C

Dr. N Chinnaiyan, G Pradeep, Vishwanath Y
Associate Professor, Information Science and Engineering Department, New Horizon College of Engineering, Bangalore, India, Chinnaiyanr@newhorizonindia.edu
Preethi J D, Beena B M,
Assistant Professor Information Science and Engineering Department, New Horizon College of Engineering, Bangalore, India. preethijd@newhorizonindia.edu

Abstract:
This Paper deals with how a string can be En-coded (compressed) & De-coded (de-compressed), so as to save the space. When a huge text or message has to be transmitted, it is encoded first and encoded message is transmitted. On the other hand, the encoded message can be decoded to get the original text or message. This is when the Huffman's coding comes into picture. This paper implements the huffmans code using c with sample data sets.
Keywords: Huffman's Coding, En-coded, De-Coded

1. INTRODUCTION

This Paper deals with how a string can be En-coded (compressed) & De-coded (de-compressed), so as to save the space. When a huge text or message has to be transmitted, it is encoded first and encoded message is transmitted. On the other hand, the encoded message can be decoded to get the original text or message. This is when the Huffman's coding comes into picture [1]. Huffman tree generated from the exact frequencies of the text "this is an example of a Huffman's tree". The frequencies and codes of each character are below. Encoding the sentence with this code requires 135 bits, as opposed to 288 (or 180) bits if 36 characters of 8 (or 5) bits were used. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.)

1.2 PROJECT DESCRIPTION

Definition:

Huffman tree generated from the exact frequencies of the text "this is an example of a Huffman's tree". The frequencies and codes of each character are below. Encoding the sentence with this code requires 135 bits, as opposed to 288 (or 180) bits if 36 characters of 8 (or 5) bits were used. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.)

Compression:

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the

weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to $n$leaf nodes and $n - 1$internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node (thus not considering them anymore), and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree.

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

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
3. Remove the two nodes of highest priority (lowest probability) from the queue
4. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
5. Add the new node to the queue.
6. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log *n*) time per insertion, and a tree with *n* leaves has 2*n*−1 nodes, this algorithm operates in O(*n* log *n*) time, where *n* is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(*n*)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
4. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
5. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
6. Enqueue the new node into the rear of the second queue.
7. The remaining node is the root node; the tree has now been generated.

Decompression

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value) [2]. Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be pre-constructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using conical encoding, the compression model can be precisely reconstructed with just $B2^B$ bits of information (where B is the number of bits per symbol). Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet). Many other techniques are possible as well. In any case, since the compressed data can include unused "trailing bits" it decompresses or must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the decompressed data along with the compression model or by defining a special code symbol to signify the end of input

1. LITERATURE SURVEY

2.1 EXISTING AND PROPOSED SYSTEM

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol). Huffman coding is such a widespread method for creating prefix codes that the term "Huffman code" is widely used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

2.2 TOOLS AND TECHNOLOGIES USED TURBO C

Turbo C was an integrated development environment (IDE) for programming in the C language. It was developed by Borland and first introduced in 1987. At the time, Turbo C was known for its compact size, comprehensive manual, fast compile speed and low price. It had many similarities to an earlier Borland product, Turbo Pascal, such as an IDE, a low price and a fast compiler, but was not as successful because of competition in the C compiler market.

Turbo C Features

Inline assembly with full access to the C language symbolic structures and names -- This allowed programmers to write some assembly language codes right into their programs without the need for a separate assembler.

Support for all memory models -- This had to do with the segmented memory architecture used by 16-bit processors of that era, where each segment was limited to 64 kilobytes (Kb). The models were called tiny, small, medium, large and huge, which determined the size of the data used by a program, as well as the size of the program itself [3]. For example, with the tiny model, both the data and the program must fit within a single 64-Kb segment. In the small model, the data and the program each used a different 64-Kb segment. So in order to create a program larger than 64 Kb or one that manipulates data larger than 64 Kb, the medium, large and huge memory models had to be used. In contrast, 32-bit processors used a flat memory model and did not have this limitation.

3. HARDWARE & SOFTWARE REQUIREMENTS

3.1 SOFTWARE CONFIGURATION

Platform : Windows 7
IDI : Turbo C

3.2 HARDWARE CONFIGURATION

System Type : INTEL
Processor : Pentium 4
Processor Speed : 2.8 GHZ
Hard Disk : 40 GB
Memory Size : 128 MB

4. SYSTEM DESIGN ARCHITECTURE

4.1 Algorithm

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.

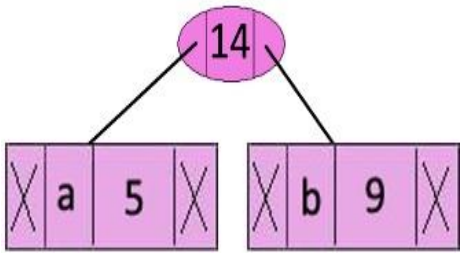
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Example

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1: Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

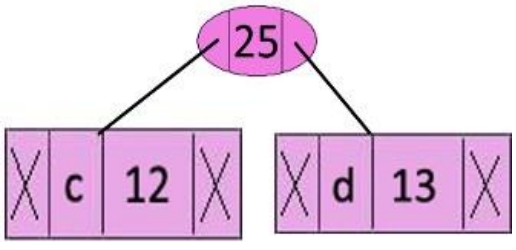
Step 2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

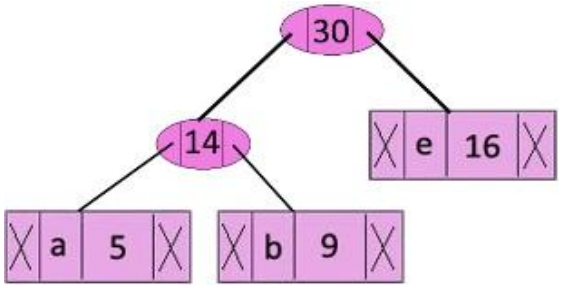
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

Character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

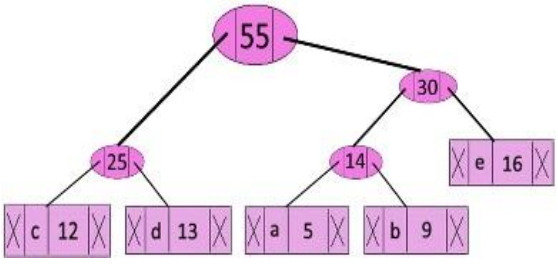
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes

character	Frequency
Internal Node	25
Internal Node	30
f	45

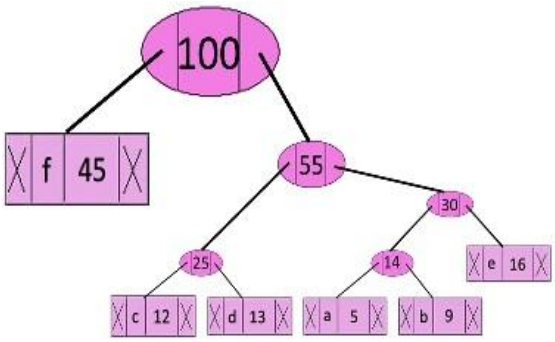
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

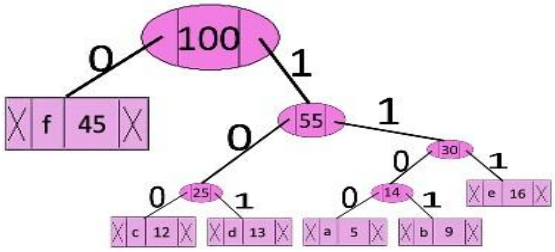


Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman's tree:
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered



The codes are as follows:

Character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

4.2 DATABASE DESIGN

ARRAYS

In C language, arrays are referred to as structured data types. An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations.

Example where arrays are used,

- to store list of Employee or Student names,
- to store marks of a students,
- or to store list of numbers or characters etc.

Advantages

1. It is used to represent multiple data items of same type by using only single name.
2. It can be used to implement other data structures like linked lists, stacks, queues, trees, graph, etc.
3. 2D arrays are used to represent matrices.

5. IMPLEMENTATION

5.1 SCREEN SHOTS

5.2 SAMPLE CODING

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define len(x) ((int)log10(x)+1)

/* Node of the huffman tree */
struct node{
int value;
char letter;
struct node *left,*right;
};
typedefstruct node Node;
/* 81 = 8.1%, 128 = 12.8% and so on. The 27th frequency is the space. Source is Wikipedia */
intenglishLetterFrequencies [27] = {81, 15, 28, 43, 128, 23, 20, 61, 71, 2, 1, 40, 24, 69, 76, 20,
1, 61, 64, 91, 28, 10, 24, 1, 20, 1, 130};
/*finds and returns the small sub-tree in the forrest*/
intfindSmaller (Node *array[], intdifferentFrom){
int smaller;
```

```
inti = 0;
while (array[i]->value==-1)
i++;
smaller=i;
if (i==differentFrom){
i++;
while (array[i]->value==-1)
i++;
smaller=i;
   }

for (i=1;i<27;i++){
if (array[i]->value==-1)
continue;
if (i==differentFrom)
continue;
if (array[i]->value<array[smaller]->value)
smaller = i;
   }
return smaller;
}

/*builds the huffman tree and returns its address by reference*/
voidbuildHuffmanTree(Node **tree){
   Node *temp;
   Node *array[27];
inti, subTrees = 27;
intsmallOne,smallTwo;
for (i=0;i<27;i++){
array[i] = malloc(sizeof(Node));
array[i]->value = englishLetterFrequencies[i];
array[i]->letter = i;
array[i]->left = NULL;
array[i]->right = NULL;
   }
while (subTrees>1){
smallOne=findSmaller(array,-1);
smallTwo=findSmaller(array,smallOne);
temp = array[smallOne];
array[smallOne] = malloc(sizeof(Node));
array[smallOne]->value=temp->value+array[smallTwo]->value;
array[smallOne]->letter=127;
array[smallOne]->left=array[smallTwo];
array[smallOne]->right=temp;
array[smallTwo]->value=-1;
subTrees--;
```

```
    }
    *tree = array[smallOne];
return;
}
/* builds the table with the bits for each letter. 1 stands for binary 0 and 2 for binary 1 (used to facilitate arithmetic)*/
voidfillTable(intcodeTable[], Node *tree, int Code){
if (tree->letter<27)
codeTable[(int)tree->letter] = Code;
else{
fillTable(codeTable, tree->left, Code*10+1);
fillTable(codeTable, tree->right, Code*10+2);
    }
return;
}
/*function to compress the input*/
voidcompressFile(FILE *input, FILE *output, intcodeTable[]){
char bit, c, x = 0;
intn,length,bitsLeft = 8;
intoriginalBits = 0, compressedBits = 0;
while ((c=fgetc(input))!=10){
originalBits++;
if (c==32){
length = len(codeTable[26]);
          n = codeTable[26];
       }
else{
length=len(codeTable[c-97]);
          n = codeTable[c-97];
       }
while (length>0){
compressedBits++;
bit = n % 10 - 1;
n /= 10;
          x = x | bit;
bitsLeft--;
length--;
if (bitsLeft==0){
fputc(x,output);
             x = 0;
bitsLeft = 8;
          }
          x = x << 1;
       }
    }
if (bitsLeft!=8){
```

```
        x = x << (bitsLeft-1);
fputc(x,output);
    }
    /*print details of compression on the screen*/
fprintf(stderr,"Original bits = %dn",originalBits*8);
fprintf(stderr,"Compressed bits = %dn",compressedBits);
fprintf(stderr,"Saved %.2f%% of      memoryn",((float)compressedBits/(originalBits*8))*100);
return;
}

/*function to decompress the input*/
voiddecompressFile (FILE *input, FILE *output, Node *tree){
    Node *current = tree;
charc,bit;
char mask = 1 << 7;
inti;
while ((c=fgetc(input))!=EOF){
for (i=0;i<8;i++){
bit = c & mask;
          c = c << 1;
if (bit==0){
current = current->left;
if (current->letter!=127){
if (current->letter==26)
fputc(32, output);
else
fputc(current->letter+97,output);
current = tree;
              }
          }
else{
current = current->right;
if (current->letter!=127){
if (current->letter==26)
fputc(32, output);
else
fputc(current->letter+97,output);
current = tree;
              }
          }
       }
    }
return;
}
/*invert the codes in codeTable2 so they can be used with mod operator by compressFile
function*/
```

```
voidinvertCodes(intcodeTable[],int codeTable2[]){
inti, n, copy;
for (i=0;i<27;i++){
      n = codeTable[i];
copy = 0;
while (n>0){
copy = copy * 10 + n %10;
n /= 10;
      }
codeTable2[i]=copy;
   }
return;
}
int main(){
   Node *tree;
intcodeTable[27], codeTable2[27];
int compress;
char filename[20];
   FILE *input, *output;
buildHuffmanTree(&tree);
fillTable(codeTable, tree, 0);
invertCodes(codeTable,codeTable2);
   /*get input details from user*/
printf("Type the name of the file to process:n");
scanf("%s",filename);
printf("Type 1 to compress and 2 to decompress:n");
scanf("%d",&compress);
input = fopen(filename, "r");
output = fopen("output.txt","w");
if (compress==1)
compressFile(input,output,codeTable2);  else
decompressFile(input,output, tree);  return 0;
```

6. SOFTWARE TESTING

Software Testing

Software testing is the process of evaluation a software item to detect differences between given input and expected output [4]. Also to assess the feature of A software item. Testing assesses the quality of the product. Software testing is a process that should be done during the development process. In other words software testing is a verification and validation process.

Verification

Verification is the process to make sure the product satisfies the conditions imposed at the start of the development phase. In other words, to make sure the product behaves the way we want it to.

Validation

Validation is the process to make sure the product satisfies the specified requirements at the end of the development phase. In other words, to make sure the product is built as per customer requirements.

Basics of software testing

There are two basics of software testing: black box testing and white box testing.

Black box Testing

Black box testing is a testing technique that ignores the internal mechanism of the system and focuses on the output generated against any input and execution of the system. It is also called functional testing.

White box Testing

White box testing is a testing technique that takes into account the internal mechanism of a system. It is also called structural testing and glass box testing. Black box testing is often used for validation and white box testing is often used for verification.

7. CONCLUSION

The main theme of algorithm is to make use of cell structures in matlab to build the Huffman tree while keeping track of child and parent nodes. Once the tree has been built, code word corresponding to each input data symbol (which acts like a leaf node in Huffman tree) can be found out by simply traversing the tree from the branch till that leaf node is encountered. The general structure contains cells corresponding to input data symbol, probability and its original order in the list of symbols passed to the algorithm as a string. Two additional cells have been added in the structure to keep information regarding the child nodes and code word of the current node. A structure is made for each data symbol and M (= number of input data symbols) instances of this structure are filled with known information and sorted in ascending order of probability. This result in M leaf nodes corresponding to M data symbols arranged in ascending order of probability.

8. BIBLIOGRAPHY

[1] https://web.stanford.edu/class/archive/cs
[2] www.springer.com/cda/content/document/cda
[3] https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/
[4] ieeexplore.ieee.org/document/7998173/