Keefer Rourke
Mr. Fryer-Davis

19 May 2015
ICS4UI

Text Compression and Decompression – Design

Running the programme

When the programme is run by the user, the text compression utility will expect some command line arguments with it. The way I expect this to work will be as follows:

```
To create a compressed file:
      huffcompress -c <input filename> <output filename>
To extract the original file from a bin file created by huffcompress:
      huffcompress -x <input filename> <output filename>
```

For ease of use, the final argument for the output filename could be optional, defaulting to an output file called <input filename>.bin, or some other logical extension in place of .bin .

If some arguments are missing, a help message should be displayed that explains how to use the programme, and then the programme should terminate so the user can start again. This is the standard behaviour for most familiar *NIX command line utilities, and so is the way it should be for consistency with the rest of the operating system.

Compressing files

If the user wants to compress a text file, the specified input file should be read, and checked for validity as a plain-text file. If the file can be opened, it should be read into a buffer, or a temporary string, so that it can be statistically analysed. A small database of the frequencies of occurrence of each character will then be created and used to build the Huffman binary tree.

The Huffman tree is created by first creating a forest of trees with a single node each. The nodes will have the structure:

```
struct node
{
    int  freq;    //the frequency that the character appears in the text
    char c;       //the character itself
    bool isLeaf;  //1 for leaf, 0 for internal node
    node * left;  //pointers to children; NULL for leaves
    node * right;
};
```

and contain the frequency of the character in the node, the character itself, whether or not it's a leaf, and pointers to it's left and right nodes. When the forest is initially created, the left and right nodes are NULL, and there is a tree for each unique character in the file that was opened.

This forest will then be made smaller, as the trees with the lowest frequency values at the root node each are combined two at time. Eventually, after $O(n\text{-}1)$ operations, this will result in a single Huffman tree, from which variable length codes for each character can be created.

The Huffman tree will effectively have two types of nodes, a leaf and an internal node. This will be denoted by the isLeaf boolean property of the node structure. The variable length prefix codes for each character will be generated by traversing the tree recursively searching for each letter until a leaf node is encountered. For every turn left, a zero is appended to the variable length prefix code, and for every

right turn, a one is appended to the prefix code.

Once the codes have been generated, they can be strung together in order of occurrence in the original file, split into 8-bit binary numbers, and saved to the disk along with a flattened representation of the Huffman tree in the compressed binary file.

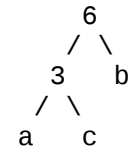
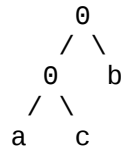
The output file
The binary file that is created will be entirely composed of 1's and 0's, and thus, is more difficult to deal with than plain-text. It's structure must be organised, consistent, and easily read without any confusion.

The file will contain:
- a binary representation of the Huffman tree that was built to encode the text
- the encoded text

Representing the Huffman tree, such that it can be built exactly as it was originally, though with less data is fairly simple matter. The frequency of each character doesn't much matter, and so does not be saved to the file. The only data that matters is the number and sequence of the nodes, which node represents which character, and whether or not the node is a leaf.

The node can be represented as an internal or leaf node by writing a 0 or 1 to the file respectively. If a 0 is encountered when parsing the file, then it represents an internal node with two children. If a 1 is encountered when parsing the file, then it represents a leaf node. Immediately following the 1, the binary version of the ASCII value for the character that is at that node is written. Once two leaf nodes have been found, jumping back to their parent node, the next child is filled out with what follows.

Using a simple example, if one were to encode the string '**aabbbc**', the tree would look like:

Huffman tree with frequencies:	Encoded tree as in the binary file:	Reconstructed Huffman tree without frequencies:
	`001a1c1b`	

Along with the encoded tree, the encoded text needs to be stored using their prefix codes to represent the letters. The prefix codes should be strung together and converted to a binary number, which is then written to the file along with the Huffman tree.

Using the above example, the prefix codes are:

a	000
b	1
c	001

So the encoded data is: `0000 0001 1100 1  (13 bits)`
And the binary file contains: `001a1c1b 0000000111001 (total 5 bytes + 2 bits)`

Keefer Rourke
Mr. Fryer-Davis

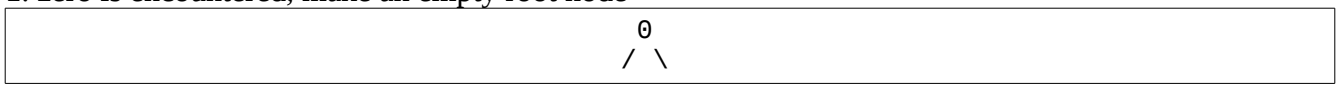
19 May 2015
ICS4UI

Decompressing files

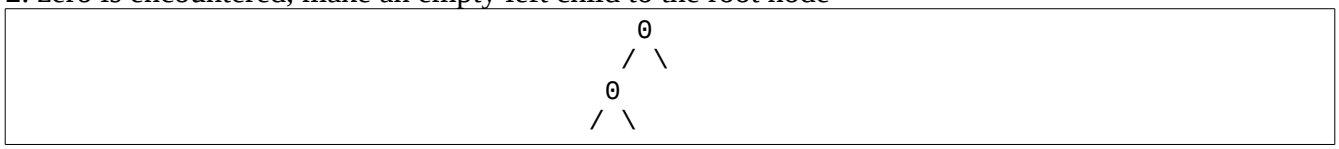
The first chunk of the compressed file will be the representation of the Huffman tree, so it must be read and a tree must be built from it. When the first 0 is encountered, the root node of the tree will be created. For every zero following, a left child will be created for the previous node until a 1 is encountered, indicating that the next node is a leaf. The leaves will be filled out left to right, filling empty children until another zero is encountered where the process repeats.

Using the example from before, the tree `001a1c1b` can be decoded as:

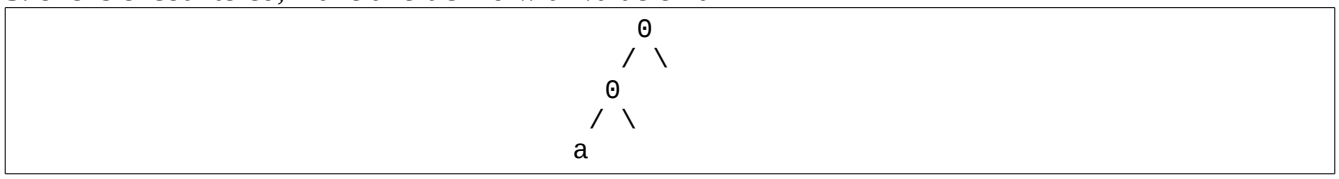
1. zero is encountered, make an empty root node



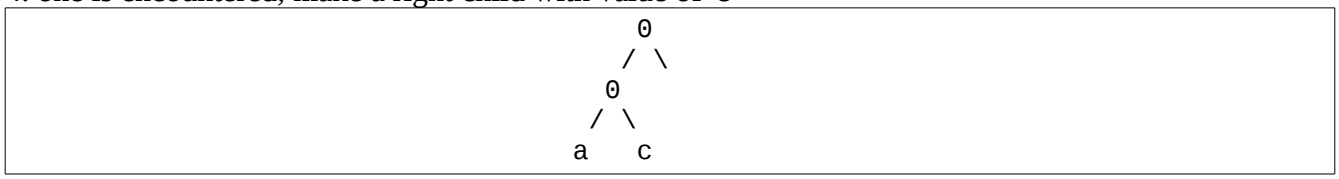
2. zero is encountered, make an empty left child to the root node



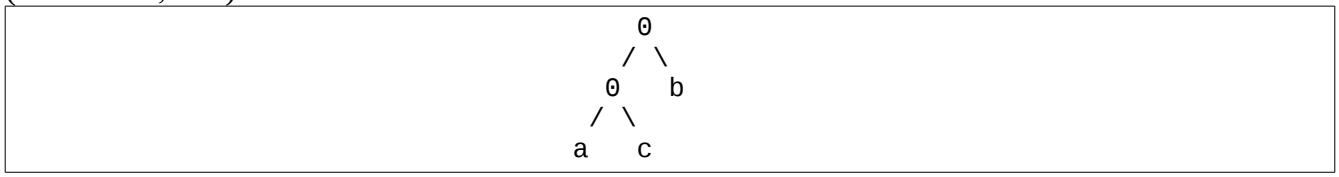
3. one is encountered, make a left child with value of 'a'



4. one is encountered, make a right child with value of 'c'



5. one is encountered, make a right child with a value of 'b' for the next node up without a right child (in this case, root)



This tree resembles the original one that was generated upon compression, and can be used to figure out what the prefix codes in the data translate to as letters. Reading the encoded data, it is used as instructions for traversing the tree. Every time a leaf node is encountered, the corresponding character should be written to a new plain-text file, and the tree traversal should start over again at the root node.

After the end of the binary file is reached, this process is terminated, and all files are closed. The new plain-text file should be exact same as the original one.