For those interested, a note on the final part of the problem, which was

And if even more time

1. Embed the Huffman Tree used for the encryption as the first bytes of the compressed file and retrieve it as part of the decompression process

In practise, the file would be compressed by one program, and uncompressed by another. The uncompression program would use the same HuffmanTree class, so that once it had the correct tree, it could use it to uncomress the file. How to convey to this program the huffman tree required?

One possibility is to embed the frequency table as a header at the beginning of the compressed file – it would have to start with an integer (delimited by a space) to say how many characters were in this table, so that these could be read (in plain text) and stored by the de-cryption program, which could then create the frequency map and from that the correct Huffman Tree. The header information would be stripped from the file before de-compression. This is probably the simplest option.

An alternative would be to store the whole Huffman tree. This tree can be represented by a pre-order traversal, which prints a 1 for every interior node visited, and a 0 for every leaf node visited. A similar pre-order traversal can be used to re-construct the tree from the sequence of 0s and 1s transmitted at the beginning of the compressed file (in fact this series could itself be compressed by representing each bunch of 8 0's and 1's as a character) How to know which character to associate with each leaf node arrived at? Again, the compressed file would have to store at the very beginning the number of characters contained in the tree, (delimited by a space) and the sequence of characters in the order they will be arrived at in the pre-order traversal.

Code for the pre-order traversal would be something like this

```
string HuffmanTree::outputTree( )

{
          char listOfChars[255];
          int noChars = 0;
          string treeRep = ""; //an empty string to start with
          repTreeRootedAt(root, treeRep, noChars, listOfChars); // start at the top
          char str[4];
          string retString(_itoa(noChars, str, 10) );
          retString.append(" "); // a space to delineate the no of chars
          retString.append(listOfChars);
          retString.append(treeRep);
          return retString;

}
```

//recursively visit sub trees

```
void HuffmanTree::repTreeRootedAt(HuffmanTreeNode *subroot,  string &treeRep,
int & noOfLeafCharsSoFar, char* listOfLeafCharsSoFar)

{
   // char* should be an array of 255, since this is the max no of discrete
chars we can have,
   // or we could size it based on previous knowledge - eg of how many chars
appeared in the frequency map
   if (subroot->isLeaf() )
   {
      treeRep.append("0");
      listOfLeafCharsSoFar[noOfLeafCharsSoFar] = subroot->value;
      noOfLeafCharsSoFar++;
   }
   else
   {
      //this is an internal node, visit it by adding a 1 to the tree
representation string, then represent its left and right sub-trees
      treeRep.append("1");
      repTreeRootedAt(subroot->left, treeRep, noOfLeafCharsSoFar,
                                                     listOfLeafCharsSoFar);
      repTreeRootedAt(subroot->right, treeRep, noOfLeafCharsSoFar,
                                                     listOfLeafCharsSoFar);
   }

}
```