ASCII TEXT COMPRESSION USING `HUFFMAN.CPP`

JACK FISHER

This project contains the following seven files:

(1) `BSTree.h`
(2) `huffman.cpp`
(3) `Huffman.h`
(4) `Makefile`
(5) `README.txt`
(6) `test.txt`
(7) `wap.txt`

The `Makefile` includes directives for compiling, using, and removing `huffman`. Useful directives include:

`make all` / `make huffman`: compiles and links the necessary files to generate the executable `huffman`
`make encode`: compresses `wap.txt` into `encoded.bin`
`make decode`: decompresses `encoded.bin` into `decoded.txt`
`make clean`: removes all files generated by the other makefile directives

To use `huffman` on your own, you may enter commands of the following form:

```
./huffman <mode> <source_file> <dest_file>
```

where `<mode>` can be either `encode` or `decode`. For `encode`, `<source_file>` must be an ASCII plaintext file. For `decode`, `<source_file>` must be a binary file created by `huffman` on the same machine. If `<dest_file>` does not exist, it will be created. If it *does* exist, if will be overwritten.

`huffman` uses Huffman encoding and Huffman decoding to compress plaintext files by constructing a Huffman tree and representing ASCII characters as bit strings of varying length.

When encoding a plaintext file, `huffman` will print a table of the characters in the file and their corresponding frequencies, followed by the Huffman tree generated from the table. It will then print another table of the characters and their corresponding bit encodings derived from the tree.

Encoding a File

To demonstrate how `huffman` works, we shall follow the compression and decompression of `test.txt`, a file provided with Project 3 for demonstration purposes. `test.txt`'s contents are as follows:

```
This is a test.
```

When `./huffman encode test.txt encoded.bin` is entered at the command line, `huffman` opens `test.txt` and reads its contents into a string. This string is then iterated over while each character's frequency is counted and compiled into a `map<char, size_t>`.[1] This map is printed as a table to the console:

```
Character frequency map:
\n      1
        3
.       1
T       1
a       1
e       1
h       1
i       2
s       3
t       2
```

The map is then used to generate a Huffman tree using the Huffman algorithm.[2] The resulting tree is printed to the console:

```
The Huffman tree (of height 5):
            (3, s)
        (5, )
                (1, .)
            (2, )
                (1, a)
    (9, )
                (1, h)
            (2, )
                (1, e)
        (4, )
            (2, t)
(16, )
                (1, T)
            (2, )
                (1, \n)
        (4, )
            (2, i)
    (7, )
        (3,  )
```

Next, the Huffman tree is used to generate the encoding for each character. Beginning at the root, traversals to a left child are recorded as a 0, and traversals to a right child are recorded as a 1. Thus, each character is guaranteed to have a unique binary encoding which is not a prefix of another character's encoding. Another map (this time a `map<char, vector<bool>>`) is created and populated to allow quick access to each character's encoding. This map is printed to the screen:

[1]In the unlikely event of a `size_t` overflow, an appropriate message is printed to the console. The program will then continue as usual, as sub-optimal compression may still be desired.

[2] See `https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html`.

```
Map of chars to bit codes:
\n      0110
        00
.       1101
T       0111
a       1100
e       1010
h       1011
i       010
s       111
t       100
```

Finally, the input string is iterated over once more, and a `vector<bool>` is constructed to represent the entire contents of `test.txt`.

Before this bitstream is written to the destination file, however, a representation of the Huffman tree used to generate the characters' binary codes must be written to the beginning of the file. This allows `huffman` to generate the tree again for decoding. The `map<char, size_t>` of character frequencies is thus encoded as an alternating sequence of `char`s and `size_t`s and terminated with the null character (`\0`).

Now the bitstream of the encoded message is converted into `char`-sized blocks (8 bits each) and written to the destination file.

A final `char` is generated with a value corresponding to the number of bits to throw out of the last `char` in the message bitstream. This `char` is written to the destination file, and the file is closed.

Decoding a File

Now that the contents of `test.txt` have been encoded as `encoded.bin`, they can be decoded with the command `./huffman decode encoded.bin decoded.txt`.

The source file is opened as before. This time its contents are read as alternating `char`s and `size_t`s, constructing the character frequency map used before. Once the null character (`\0`) is read, the map is complete. The Huffman tree is derived from the map as before, and the tree is printed to the screen. As long as `encoded.bin` is encoded and decoded on the same machine, the derived tree will be the identical to the one used for encoding.

The tree is used as before to generate a map of characters and their respective codes, and the remainder of the file is read as a series of `char`s which are interpreted bit-by-bit in a `vector<bool>`. When the final `char` is read, its value is added to its own size (8 bits), and the resultant sum $8+n$ is used to remove the last $8+n$ bits from the `vector<bool>`. These n bits represent the extra space required to make a complete char when `encoded.bin` was written.

Finally, the `vector<bool>` is iterated through, generating the appropriate chars which are written to the destination file before the file is closed.

A Note on Compression

You may notice that the size of `test.txt` is 16 bytes while the size of `encoded.bin` is 99 bytes (over 600% of the original size). This is due to the encoding of the Huffman tree within the file — which, for small files, outweighs any compression gained.

Compressing the included `wap.txt` (3,348,584 bytes) will yield an encoded file of 1,892,343 bytes — 57% of the original file size.

The greatest compression will be achieved with large source files containing small subsets of the characters in the ASCII alphabet.