# Tutorial 6: Huffman Encoding

Computer Science 244, Stellenbosch University

Due: 23 October 2017

## Contents

## List of Algorithms

## 1 Overview

Data compression forms an integral part of everyday computing, and many compression algorithms have been developed to cope with specific data sets. Compression algorithms are either *lossless*, allowing the exact original data to be reconstructed from the compressed data, or *lossy*, where in return for better compression rates, only an approximation of the original data can be reconstructed. Lossy algorithms, found in JPEG (for images) and MPEG (for audiovisual data), are commonly used for multimedia data; lossless algorithms, such as those in GZIP and BZIP2, are used for textual and non-multimedia data. For this tutorial, you will complete a lossless file compression utility for Linux.

```
Require: $S$, a string of characters.
Ensure: $T$, a Huffman tree for $S$.
 1: Insert all the characters in $S$ into a heap $H$ according to their frequencies
 2: while $H$ is not empty do
 3:     if $H$ contains only one entry $x$ then
 4:         make $x$ the root of $T$
 5:         remove $x$ from $H$
 6:     else
 7:         remove two entries $x$ and $y$ from $H$
 8:         create a new entry $z$, and assign $x$ and $y$ as its children
 9:         $z$.frequency $\leftarrow$ $x$.frequency + $y$.frequency
10:         insert $z$ into $H$
11:     end if
12: end while
```

Algorithm 1: Huffman Tree Construction

## 2 Huffman Codes

Huffman coding, developed by David Albert Huffman (1925–1999) while he was a doctoral student at MIT, is an efficient method to compress an arbitrary string of characters, reducing the amount of storage it requires. The encoding uses a data structure called a *Huffman tree* that generates unique encodings for different characters based on their frequencies. Characters that appear more often in the uncompressed string are assigned short encodings, whereas less frequent characters are assigned longer encodings. Once a Huffman tree has been constructed, every character in the original input string is replaced with its encoding derived from the Huffman tree.

### 2.1 Algorithms

The algorithms for Huffman tree construction, heap insertion, and heap removal are given as Algorithms 1 to 3. Note that we use an array representation for the binary tree on which the heap is built. Refer to your Computer Science 214 text book for more information on heaps and their properties, as well as the array representation of binary trees.

### 2.2 An Example

Consider a file that contains the following sequence of characters:

```
EFBEAAEBAEAAECCEDDEDDECEE
```

A frequency table is constructed to record the number of times a specific character occurs. Remember that there are 256 possible characters because every character is stored in one byte (eight bits). The frequency table is given as Table 1.

Next, a heap containing the characters is constructed by moving through the table and inserting an item into the heap if the frequency of the character is greater than zero. For our example, the complete heap will look as follows:

| **Heap Index:** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Heap Entry (char, freq):** | (F, 1) | (D, 4) | (B, 2) | (A, 5) | (E, 10) | (C, 3) |

```
Require: A heap $H$, and an element node $n$ to insert.
Ensure: The heap $H$ containing the inserted element.
 1: $H.\mathsf{a}[H.\mathsf{n}] \leftarrow n$
 2: $c \leftarrow H.\mathsf{n}$                         ▷ $c$ is the index of the child
 3: $p \leftarrow (c - 1)/2$                       ▷ $p$ is the index of the parent
 4: while $p \geq 0$ do
 5:     if $H.\mathsf{a}[p] > H.\mathsf{a}[c]$ then
 6:         swap $H.\mathsf{a}[p]$ and $H.\mathsf{a}[c]$
 7:         $c \leftarrow p$
 8:         $p \leftarrow (c - 1)/2$
 9:     else
10:         $p \leftarrow -1$
11:     end if
12: end while
13: $H.\mathsf{n} \leftarrow H.\mathsf{n} + 1$
```

Algorithm 2: Heap Insertion

```
Require: A heap $H$.
Ensure: The heap $H$ with the minimum element $m$ removed.
 1: if $H.\mathsf{n} \geq 0$ then
 2:     $m \leftarrow H.\mathsf{a}[0]$
 3:     $H.\mathsf{a}[0] \leftarrow H.\mathsf{a}[H.\mathsf{n} - 1]$
 4:     $H.\mathsf{n} \leftarrow H.\mathsf{n} - 1$
 5:     $p \leftarrow 0$                            ▷ $p$ is the index of the parent
 6:     $c \leftarrow 2p + 1$                        ▷ $c$ is the index of the child
 7:     while $c \leq H.\mathsf{n} - 1$ do
 8:         if $H.\mathsf{a}[c] \geq H.\mathsf{a}[c + 1]$ then
 9:             $c \leftarrow c + 1$
10:         end if
11:         if $H.\mathsf{a}[c] \leq H.\mathsf{a}[p]$ then
12:             swap $H.\mathsf{a}[c]$ and $H.\mathsf{a}[p]$
13:             $p \leftarrow c$
14:             $c \leftarrow 2p + 1$
15:         else
16:             $c \leftarrow H.\mathsf{n}$
17:         end if
18:     end while
19: end if
```

Algorithm 3: Heap Removal

Table 1: A frequency table for the example in Section 2.2.

| Table Index | Character | Frequency | Table Index | Character | Frequency |
|---|---|---|---|---|---|
| 65 | A | 5 | 68 | D | 4 |
| 66 | B | 2 | 69 | E | 10 |
| 67 | C | 3 | 70 | F | 1 |

The bottom-up construction of a Huffman tree is illustrated in Figures 1 to 5. A table that can be used for encoding can now be built by traversing the Huffman tree in Figure 5. Note that all of the original characters are located at the leaves of the tree. We record the path to every leaf as a sequence of bits—`0` for a left traversal and `1` for a right traversal. The sequence of bits tracing a path from the root to any particular leaf becomes the new encoding for the character located at that leaf.

For example, `E`, the most frequent character, will be encoded as `0`, while `D` will be encoded as `110`. Since `F` and `B` appear least frequently in the original input file, each has the greatest encoding length of four bits, but, in this example, still shorter than the original encoding length of eight bits.

Once the encoding table has been constructed, the output file must be generated. This is done by reading a character from the input file, locating its entry in the encoding table, and writing its Huffman code, i.e., the sequence of bits in its new encoding, to the output file. Since at least one byte must be written to the output file, we must be careful when the encoding uses fewer than eight bits: In some cases, the Huffman code for a specific character overflows into a second byte in the output file.

Consider the original sequence of characters in the input file. To encode the first `E`, we write `0` to bit 7 of the output byte so that it contains `0`—. Then we write the encoding of the `F` as `1010`, so that the byte contains `01010`—. The `B` requires four bits for its encoding `1011`, so that the first byte contains `01010101`, and the second byte contains `1`—.

## 3 Specification

The compression application is divided into three units:

1. the functions to construct the heap used during Huffman encoding,
2. the functions to construct a Huffman tree and to build an encoding table, and
3. the application that uses these function to perform Huffman compression of an arbitrary input file.

### 3.1 Design

Items 1 and 2 must be implemented in 32-bit Intel assembly language, and item 3 as a C language program. The functions specified by name in the following sections *must be present* in your final implementation. You may add additional functions to test your code, but you must not deviate from the interface given here.[1]

The C data structures required by the functions are given in Figures 6 and 7. You must use these structures exactly as given, otherwise it will not be possible to test your functions separately. The type definitions in Figures 6 and 7 are available in the header files `heap.h` and `huffman.h`, respectively.

### 3.2 Heap Manipulation

All routines related to the initialisation, construction, and manipulation of the heap structure must be written in the `heap.asm` file. The following routines must be implemented:

[1]In this section, as well as those following, words like "must", "may", and "should" are used to signify what is required. Refer to RFC2119 (`http://www.faqs.org/rfcs/rfc2119.html`) for definitions and discussion.

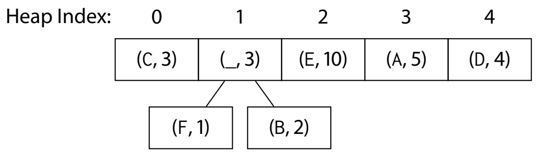

Figure 1: The heap contains more than one entry, so F and B are removed because they have the lowest frequencies. A new entry, with a frequency of 3 = 1+2 and that contains F and B as its left and right child, respectively, is created. Finally, this new entry is inserted into the heap.

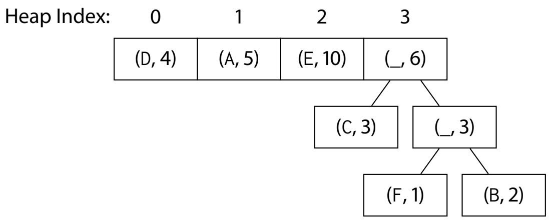

Figure 2: The next two entries with the lowest frequencies are now removed, that is, C and the entry that was created in the previous step. These two entries are removed from the heap. A new entry with a frequency of 6 = 3 + 3 is created and inserted into the heap.

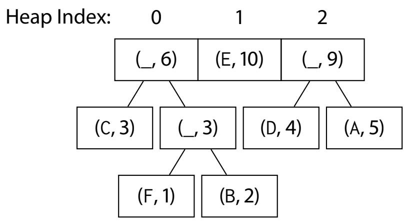

Figure 3: The entries D and A are now removed. A new entry with a frequency of 9 = 4+5 and with D and A as its children is created and inserted into the heap.

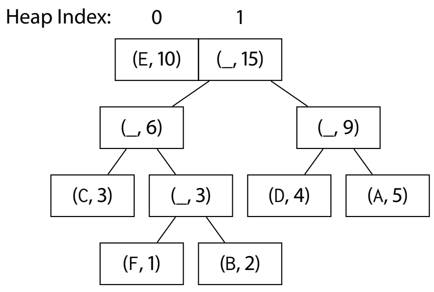

Figure 4: The entries (_, 6) and (_, 9) are now removed. A new entry with a frequency of 15 is created and inserted into the heap.

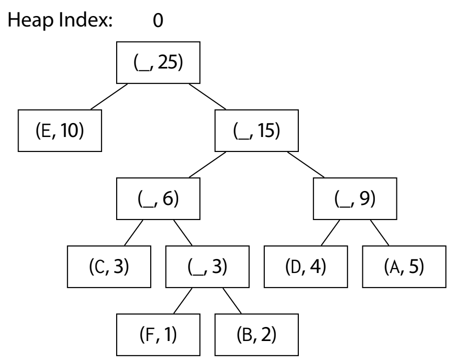

Figure 5: The last two entries are now removed and added as children to a new entry, which has a frequency of 25 = 15 + 10 and is inserted into the heap. On the next iteration, the algorithm will return this node as the root of the Huffman tree.

```
#define MAX_HEAP_SIZE 256

typedef struct heap_node HeapNode;
struct heap_node {
    int frequency;
    char c;
    HeapNode *left;
    HeapNode *right;
};

typedef struct heap {
    int n;
    HeapNode a[MAX_HEAP_SIZE];
} Heap;
```

Figure 6: C data structures required to handle the heap.

```
typedef struct huffman_node {
    int huffman_code;
    int bit_size;
} HuffmanNode;
```

Figure 7: C data structures required to handle the Huffman tree and lookup table.

- `void heap_initialize(Heap *H)`
  initialises the heap structure `H` by setting the heap count to zero, and assigns default "zero" values to every entry inside the heap.
- `void heap_insert(Heap *H, HeapNode *node)`
  inserts a new entry `node` into the heap `H`.
- `void heap_remove(Heap *H, HeapNode *node)`
  removes the top entry node from the heap `H`, reorganises the heap, and returns the removed entry in `node`.

### 3.3 Huffman Tree Construction

The functions associated with the Huffman tree are contained in the `huffman.asm` file as 32-bit assembly language code. The following three functions, specified in the header file `huffman.h`, must be implemented:

- `void huffman_build_tree(Heap *h, HeapNode **t)`
  constructs a Huffman tree from the heap.
- `void huffman_initialize_table(HuffmanNode *t)`
  initialises a table to hold the Huffman codes for individual characters.

-
  ```
  void huffman_build_table(HeapNode *root, HuffmanNode *t,
                           int code, int size)
  ```
  creates a table containing 256 entries by traversing the Huffman tree (recursively) in-order; the current encoding—i.e., the path already traced—is contained in `code` and its length in `size`, and each entry inside the table contains the Huffman code and the number of bits required to store the code.

### 3.4 The Main Program

The `main` function must be implemented in the `main.c` and is responsible for the following tasks:

- Report and exit with a non-zero exit code if the first command-line argument cannot be opened as a file.
- Open the input file, count the character frequencies, and call the appropriate functions to build the heap and create the corresponding Huffman tree.
- Reopen (or rewind) the input file, and then create the output file, starting with the header, followed by the Huffman-encoded data.

### 3.5 The Output File

The output file generated by the utility must receive "`.cz`" as extension. For example, the command invocation "`./compress myfile.txt`" must produce a compressed file called "`myfile.txt.cz`".

The output file must contain a header that can be used during decompression to obtain the Huffman tree that was used when the original file was compressed. Although it is possible to generate a compressed Huffman tree as header, you are not required to do so. The header of the output file must contain the following information:

- The first four bytes contain an integer value that specifies the number of characters in the input file with non-zero frequencies.
- This entry is followed by all the frequency entries. Each entry consists of 5 bytes: 1 `char` (1 byte), followed by 1 `int` (4 bytes). The entries must be ordered in ascending UTF-8 values, for example: (`A`, 300), (`F`, 20), (`b`, 14), etc. The encoded byte stream immediately follows the header.

## 3.6 Restrictions

You may use any of the standard C library functions contained in `assert.h`, `ctype.h`, `errno.h`, `float.h`, `limits.h`, `locale.h`, `math.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`. You may not use any other libraries, except those that you write yourself. Note, in particular, that you must use `malloc` (or its associated allocation functions in `stdlib.h`) for memory management.

## 3.7 Style

Use appropriate `%define` directives in your assembly language source files to define magic numbers such as the size of and the maximum number of entries in the heap, the size of the Huffman table, and the sizes of different types of node. For example, the size of the heap should be defined in terms of the maximum number of heap entries and the size of an individual node. You may use any additional functionality provided NASM, even if not explicitly covered in the course. In particular, you may use macros *as long is you write such macros yourself.*

## 3.8 Some Pointers

(A very bad pun, I admit.)

1. Although you will probably start by coding a function in C and then reimplementing it in assembly language, your goal should be to think in assembly. Especially where double pointers and pointers to structures are involved, the C syntax can be quite involved, but the assembly almost surprisingly simple. Also, when you have to initialise memory, which typically means filling it with zeros, you should do the equivalent of a `memset`[2] in assembly language with a repeated instruction.
2. The Makefile in your tutorial repository contains rules for both C and assembly language implementations. Since assembly language is difficult to test, the idea is that you can mix and match C and assembly implementations as you go on.
3. All of this said, it is probably best if you start by making a quick C implementation. After the compiler project, the simple heap and Huffman algorithms should not be a problem. However, the encoding in the main program can be quite tricky, so allocate enough time for that. Once you have C implementations, you should start replacing the required functions one-by-one.

[2] Do `man memset` on the command line.

## 4 Marking scheme

This tutorial is due on Monday, 23 October 2017, at 09:00. There are no explicit style marks, but up to 10% of the total available marks may be subtracted for a disorganised implementation. There will also not be any caps for segmentation errors, but your work may be penalised if it compiles or assembles with warnings.

Very important, however, is that your tutorial code must be built with the provided Makefile, otherwise your work cannot be tested. You may reorganise the `C_OBJS` and `ASM_OBJS` rules to reflect what you have completed, but that is it.

| DESCRIPTION | WEIGHT |
|---|---|
| Heap manipulation | 35 |
| Huffman tree construction | 35 |
| Main program | 30 |

Please note the following:

- The heap manipulation and Huffman tree construction can be tested separately *only if you keep to the specification.*
- The main program will not be marked separately, without the other two units. To qualify for a distinction, you must implement all three units.
- If you run out of time with the assembly, *but you have a working implementation in C for all units*, you may submit either (but not both) of the heap manipulation or Huffman tree construction units in C. But in this case, your marks for that C implementation will be out of 10 instead of 35, which means you can only get a distinction if everything works perfectly. *If you submit everything in C, you will get zero for the tutorial.*
- Your `compress` program will only be tested on text files, but these may contain UTF-8 characters. You don't have to do anything special, but don't assume that only a subset of the byte patterns will be used.
- A binary `uncompress` program is available in your repository for testing.
- Your repository is at `rw244-2017@git.cs.sun.ac.za:`⟨US number⟩`/tut06`

## References

[1] Knuth, Donald E. 1997. *Fundamental Algorithms*. Volume 1 of *The Art of Computer Programming*. Boston, Mass.: Addison-Wesley.

[2] Salomon, David. 2008. *A Concise Introduction to Data Compression*. London: Springer-Verlag.

[3] Salomon, David. 2007. *Data Compression: The Complete Reference*. Fourth edition. London: Springer-Verlag.