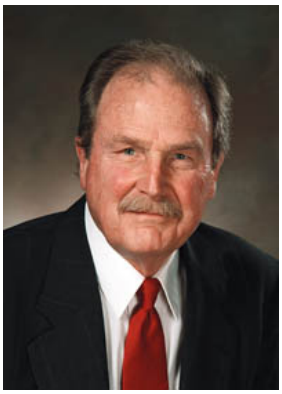
Assignment 4 — Data Compress
Huffman Trees, Stacks, and Priority Queues

Prof. Darrell Long
Department of Computer Engineering

Spring 2017



1 Introduction

> *You're trying to take something that can be described in many, many sentences and pages of prose, but you can convert it into a couple lines of poetry and you still get the essence, so that's compression. The best code is poetry.*
>
> —Satya Nadella

When David Huffman was a graduate student in a class at MIT, the professor gave the class an unsolved problem: How to construct an optimal static encoding of information. The young Huffman came back a few days later with his solution, and that solution changed the world. Data compression is now used in all aspects of communication. David Huffman joined the faculty of MIT in 1953, and in 1967 he joined the faculty of University of California, Santa Cruz as one of its earliest members and helped to found its Computer Science Department, where he served as chairman from 1970 to 1973. He retired in 1994, and passed away in 1999.

The key idea is called *entropy*, originally defined by Claude Shannon in 1948. Entropy is a measure of the amount of information in a, say, set of symbols. If we define $I(x) = \log_2 \Pr[x]$ to be the information content of a symbol, then the entropy of the set $X = \{x_1, \ldots, x_n\}$ is

$$H(X) = \sum_{i=1}^{n} \Pr[x_i] I(x_i) = -\sum_{i=1}^{n} \Pr[x_i] \log_2 \Pr[x_i].$$

It should be easy to see that the optimal *static* encoding will assign the least number of *bits* to the most common symbol, and the greatest number of bits to the least common symbol.

Your task will be to find a Huffman encoding for the contents of a file, and use it to compress that file. You must also be able to reconstruct the original file from its compressed encoding.

In order to do this, you will need to:

1. Compute a histogram of the file, in other words, count the number of occurrences of each byte in the file.
2. Construct the Huffman tree that represents this histogram, in order to do this you will use a *priority queue*.
3. Construct the code for each symbol, in order to do this you will use a *stack* and perform a traversal of the Huffman tree.
4. Emit an encoding of the Huffman tree to a file, in order to do this you will perform a *post-order* traversal of the Huffman tree.
5. Emit an encoding of the original file to the compressed file, in order to do this you will use *bit vectors* with two additional operations: `append` and optionally `concatenate`.
6. Read the tree from the compressed file, in order to do this you will use a *stack*.
7. Decode the compressed bit stream into an identical copy of the original file. In order to do this, you will use the bits (0 means left, 1 means right) to guide your traversal of a reconstructed Huffman tree.

1.1 Stacks

> *I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*
>
> —Linus Torvalds

You will be using stacks in at least two instances in this assignment, consequently you will need several different types of nodes. You may, for example, want a stack of `treeNode` and a stack of `uint32_t`, or even better a stack of *bits* if you are trying to be extra clever (so, let me give you some code for that).

Below is the generic interface for a stack.

```
# ifndef _STACK_H
# define _STACK_H
# include <stdint.h>
# include <stdbool.h>

typedef uint32_t item; // You will need to change this

typedef struct stack
{
    uint32_t size; // How big?
    uint32_t top;  // Where's the top?
    item *entries; // Array to hold it (via calloc)
} stack;

stack *newStack();         // Constructor
void   delStack(stack *); // Destructor

item pop (stack *);        // Returns the top item
void push(stack *, item); // Adds an item to the top

bool empty(stack *);       // Is it empty?
bool full (stack *);       // Is it full?
```

```
23
24 # endif
```

stack.h

The code for each type of stack will be identical, except for the type of items being operated on. Unlike some languages with *templates* or *generics*, you will need to have code for each type. You could try to be exceedingly clever and use `(void *)`, but again, I caution you against that approach.

1.2 Priority Queues

> *Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.*
>
> —Rob Pike

In order to construct the Huffman tree, you will want to use a data structure called a *priority queue*. A priority queue is like a regular queue in that you remove items from the *tail*, but differs in that when you remove an item it is always the *smallest* (or largest) item. This implies that the `enqueue` operation does not simply insert at the *head*. Of course, the `dequeue` operation could *search* for the smallest item each time, but that is a *bad idea*. Note that the definition does not say what the order in the rest of the queue must be, only that the `dequeue` operation will return the smallest item.

```
1  # ifndef _QUEUE_H
2  # define _QUEUE_H
3  # include <stdint.h>
4  # include <stdbool.h>
5
6  typedef treeNode item; // treeNode defined in huffman.h
7
8  typedef struct queue
9  {
10     uint32_t size;       // How big is it?
11     uint32_t head, tail; // Front and rear locations
12     item *Q;             // Array to hold it (via calloc)
13 } queue;
14
15 queue *newQueue(uint32_t size); // Constructor
16 void   delQueue(queue *q);      // Destructor
17
18 bool empty(queue *q);           // Is it empty?
19 bool full (queue *q);           // Is it full?
20
21 bool enqueue(queue *q, item  i); // Add an item
22 bool dequeue(queue *q, item *i); // Remove from the rear
23
24 # endif
```

queue.h

1.3 Huffman Trees

> *It's easy to make mistakes that only come out much later, after you've already implemented a lot of code. You'll realize "Oh I should have used a different type of data structure." Start over from scratch.*
>
> —Guido van Rossum

A *Huffman tree* is a *full binary tree* where the leaves represent the symbols to be encoded, and the interior nodes provide a path from the root to the leaves. The path is labeled with 0 for left, and 1 for right. The most common symbol (the one with the highest count in the histogram) should have the shortest path; the least common symbol the longest path.

The process of creating a Huffman tree is surprisingly simple. First, compute your histogram. Next, using `treeNode *newNode(uint8_t s, bool l, uint64_t c)` enqueue a node for each entry in the histogram into the *priority queue*.

Third, repeat until the priority queue is empty: dequeue two nodes from the priority queue. Join them under a new node, and insert that node into the priority queue. If there is only *one* node remaining in the priority queue, then that is the root of the Huffman tree.

```
# ifndef _HUFFMAN_H
# define _HUFFMAN_H
# include <stdint.h>
# include <stdbool.h>

# ifndef NIL
# define NIL (void *) 0
# endif

typedef struct DAH treeNode;

struct DAH
{
    uint8_t  symbol;
    uint64_t count;
    bool     leaf;
    treeNode *left, *right;
};

// New node, with symbols, leaf or not, a count associated with it
treeNode *newNode(uint8_t s, bool l, uint64_t c);

// Dump a Huffman tree onto a file
void dumpTree(treeNode *t, int file);

// Build a tree from the saved tree
treeNode *loadTree(uint8_t savedTree[], uint16_t treeBytes);

// Step through a tree following the code
int32_t *stepTree(treeNode *root, treeNode *t, uint32_t code);

// Parse a Huffman tree to build codes
void buildCode(treeNode *t, code s, code table[256]);
```

```
34
35 // Delete a tree
36 void *delTree(treeNode *t);
37
38 static inline void delNode(treeNode *h) { free(h); return; }
39
40 static inline int8_t compare(treeNode *l, treeNode *r)
41 {
42     return l->count - r->count; // l < r if negative, l = r if 0, ...
43 }
44
45 treeNode *join(treeNode *l, treeNode *r); // Join two subtrees
46 # endif
```

huffman.h

The function `treeNode *join(treeNode *l, treeNode *r)` takes two nodes, which may be either leaves or interior nodes, and creates a new internal node with its count set to the sum of the counts of the two child nodes.

When you are creating the code for each symbol, you will want to use a stack as defined in `code.h`, it provides a lovely little stack of bits. You will see that as you traverse the Huffman tree, you want to push `0` when you go to the left child and push `1` when you go to the right. Consequently, when you return from either child, you will want to perform a pop. As soon as you reach any leaf node during the traversal, the current state of the stack will represent the code for the symbol at the leaf node.

```
1  # ifndef _CODE_H
2  # define _CODE_H
3
4  # include <stdint.h>
5  # include <stdbool.h>
6
7  typedef struct code
8  {
9      uint8_t bits[32];
10     uint32_t l;
11 } code;
12
13 static inline code newCode()
14 {
15     code t;
16     for (int i = 0; i < 32; i += 1) { t.bits[i] = 0; }
17     t.l = 0;
18     return t;
19 }
20
21 static inline bool pushCode(code *c, uint32_t k)
22 {
23     if (c->l > 256)
24     {
25         return false;
26     }
27     else if (k == 0)
```

```
28      {
29          c->bits[c->l / 8] &= ~(0x1 << (c->l % 8));
30          c->l += 1;
31      }
32      else
33      {
34          c->bits[c->l / 8] |=  (0x1 << (c->l % 8));
35          c->l += 1;
36      }
37      return true;
38  }
39
40  static inline bool popCode(code *c, uint32_t *k)
41  {
42      if (c->l == 0)
43      {
44          return false;
45      }
46      else
47      {
48          c->l -= 1;
49          *k = ((0x1 << (c->l % 8)) & c->bits[c->l / 8]) >> (c->l % 8);
50          return true;
51      }
52  }
53
54  static inline bool emptyCode(code *c) { return c->l ==   0; }
55
56  static inline bool fullCode (code *c) { return c->l == 256; }
57  # endif
```

code.h

You may find it convenient to create a function `appendCode` for a `bitVector` which can take in a `code *` and append that code to the bit vector.

2 Your Task

You should construct both an *encoder* and a *decoder*. The encoder takes *any* file and produces a compressed file. The decoder takes a compressed file and produces an *exact duplicate* of the original file, down to each bit.

3 Specifics

You will be pulling together most of the data structures that you learned this quarter to finish this assignment. You will need at least two different stacks, a priority queue, arrays, and of course, the Huffman tree.

3.1 Encoding

Encoding is the concept of taking a source file and compressing it to reduce its size. For this section, I will refer to the source file as `sFile` and the output (compressed) file as `oFile`. To encode an `sFile`, follow these steps:

1. Open the `sFile`, and read through it to construct your histogram. Your histogram could be a simple array of 256 `uint32_t`'s (because a byte can only hold 256 different values).

2. Increment the count of element 0 and element 255 by one in the histogram. This is so that at the very minimum, the histogram will have two elements present. Do this regardless of what you read in. While doing this may result in a non-optimal Huffman Tree later on, it is a quick and clean solution to handling the case when a file has no bytes or has bytes of the same value.

3. For each entry in the histogram where the count is greater than 0 (there should be at minimum two elements because of step 2), create a corresponding `treeNode` and insert this node into the priority queue.

4. Use the priority queue to construct the Huffman tree. You do this by acquiring the two smallest elements in the queue, adding their count together and creating an internal node (`leaf == 0`, `symbol == $`). You then join the two elements you initially acquired as the parents of this new node using `treeNode *join`. Then you insert this new node back into the priority queue. You can use any symbol you like to represent an internal node, but I urge you to be consistent.

5. Perform a *post-order traversal* of the Huffman tree (`buildCode`).

   (a) If the current node is a leaf, the current stack (`code s`) represents the path to the node, and so is the code for it. Save this stack into a table of variable length codes corresponding to each symbol (`code table[256]`).

   (b) If it is an interior node, `push(0)`, and follow the left link.

   (c) After you return from the link, `pop()` the stack, and `push(1)`, and follow the right link.

6. Write out a `uint32_t` magic number onto the `oFile`. This number is `0xdeadd00d`. This magic number identifies a file as one which has been compressed using your encoder. It is crucial that you use this magic number and nothing else.

7. Write the length of the original file (in bytes) to the `oFile` as a `uint64_t`. This will help you debug when performing decoding and also allows you to acquire the size of the array you will need when constructing the original file from the compressed file.

8. Write out the size of your tree (in bytes) to the `oFile` as a `uint16_t`. This size will be `(3 * leafCount) - 1` (but never less than *zero*).

9. Perform a *post-order traversal* of the Huffman tree to write the tree to the `oFile`. This should be a function called `dumpTree` and should write `L` followed by the byte of the symbol for each leaf, and `I` for interior nodes. You should not write a symbol for an interior node.

10. Beginning at the start of the `sFile`, for each symbol copy the bits of the code for that symbol to the `oFile`. It may prove easier to append these bits to a long bit vector, and then write the bytes of the bit vector.

11. Close both files.

12. Make sure that you follow the order for creating the `oFile`. If you do not, it will not work.

    (a) Magic number (`uint32_t`): `0xdeadd00d`.

    (b) Size of original file (`uint64_t`).

    (c) Size of Huffman tree (`uint16_t`): `(3 * number of leaf nodes) - 1`.

    (d) The Huffman tree using a *post-order* traversal.

    (e) The encoding of the original file.

3.2 Decoding

Decoding is the concept of taking a compressed file and expanding it to match the original file. For this section, I will refer to the source (compressed) file as `sFile` and the uncompressed file as `oFile`. To decode a file, follow these steps:

1. Read in the magic number which should be the first 4 bytes of the `sFile`. In case this magic number does not match `0xdeadd00d`, then an invalid `sFile` was passed into your program. Display a helpful error message and quit.

2. Read in the next 8 bytes of the `sFile`. This should give you the exact size of `oFile`. Your file must be an exact copy of the original uncompressed file, so it cannot be any longer or any shorter. You can use this size to help you in debugging during the entire decoding phase.

3. Read in the next 2 bytes of the `sFile` and call this `treeSize`. Next, allocate an array (`savedTree`) of `uint8_t`'s which is `treeSize` long. Read in the `sFile` for `treeSize` bytes into `savedTree`. This loads all the binary information of the tree into the array and you can use this array to reconstruct your Huffman tree.

4. Reconstruct the Huffman tree using `loadTree`. You should use a stack to reconstruct the tree (recall it was written in *post-order*, so it is: *leaf*, *leaf*, *parent*).

   (a) Iterate over the contents of `savedTree` from 0 to `treeSize`.

   (b) If the element of the array is an `L`, then the next element will be the symbol for the leaf node. Use that symbol to create a node using `newNode`. Now, `push` this new node back onto the stack.

   (c) If the element of the array is an `I`, then you have encountered an interior node. At this point, you `pop` once to get the right child of the interior child and then `pop` again to acquire the left child. Now, create the interior node using `join` and then `push` the interior node back into the stack.

   (d) After you finish iterating the loop, `pop` one last time. This should give you back the root of your Huffman tree.

5. Now, begin reading in a bit at a time from the `sFile`. You may do this by reading a single bit at a time or by reading the entire bit stream into a bit vector. For each bit you read, step through the tree using `stepTree`.

   (a) Begin at the root of the Huffman tree. If a bit of value 0 is read, then move into the left child of the tree. If a bit of 1 is read, then move into the right child of the tree.

   (b) In case after stepping you are at a leaf node, then return the symbol for that leaf node and reset your state to be back at the root. Output this symbol onto the `oFile`. *Note*: You may buffer these symbols into an array and then write out the entire array once at the end. *Hint*: The size of this array should be known to you because of step 2.

   (c) In case after stepping you are at an interior node, then simply return -1, signifying that a leaf node has not yet been reached.

   (d) Repeat until all bits in the `sFile` have been exhausted (*Caution*: there may very well be a few extra bits in the last byte, and those could make a symbol; emitting that symbol would be *wrong*).

6. At this point, you should have a fully decompressed `oFile` which should exactly match the size of the original file.

7. Close both files.

4 Expectations

This project will be completed by groups of size *at most two.* That means you can pick a parter, or do it alone. This is *not pair programming!*

- Partner_1 will be responsible for the encoder.
- Partner_2 will be responsible for the decoder.
- Both partners will collaborate to develop the relevant data structures.
- Both partners will submit *exactly* the same code.
- Each partner will submit their own `README` file.
  - Describe the design.
  - Detail their contribution.
  - Detail their partner's contribution.

5 Deliverables

You need to submit the usual items:

1. `Makefile`
2. `encode.c`
3. `decode.c`
4. `huffman.c` and `huffman.h`
5. Any other files that are required to build your program.

- Your encoding program *must* be called `encode`.
- Your decoding program *must* be called `decode`.
- Both must accept the `-i inputFile` option. This option is *not optional* – in other words, you must specify an input file.
- Both must accept the `-o outputFile` option. This option is *optional* – in other words, it can write to `stdout`.
- Both must accept the `-v` flag to turn on verbose mode. This mode should print helpful information regarding the encoding/decoding process (Eg: `treeSize`, size of compressed file, etc).

6 Strategy

Let's talk strategy.

First, develop the data structures that you will need. Draw pictures, work them out, implement them, and test them. Test them again.

Second, work out the steps (we've told them to you already, but there may be little details). Draw a diagram, think it through, and start putting the pieces together.

For example, make a histogram. Try it. Does it work? Good. Make a Huffman tree node, based on an entry in the histogram. Did that work? Good. Insert them one by one into the priority queue. Pull one off the priority queue. Is it the smallest one? Good. Try the next one. Did that work? Good. Write the `join` function. Test it. ...

There will be working versions of `encode` and `decode` in Professor Long's directory, and a file called `secret.zzZ` that you can try to decode.