Compressing Files with Huffman Encoding

Moore Macauley
University of Santa Cruz, California

November 25, 2022

1 Introduction

For my final assignment, I will be compressing text files using Huffman Encoding. The general idea of how this works is fairly simple. We create a tree of characters, where frequent characters are closer to the root, and uncommon characters are further away. We then create codes consisting of binary, where a 0 means to go down the tree to the left, and a 1 means go down the tree to the right. Once a leaf is reached, that code represents the character. This allows common characters to be represented with much fewer than 8 bits, allowing any text document to become much smaller.

2 Nodes

In order to do this, we must first create the nodes that will be contained inside the huffman tree. These nodes will contain a pointer to the node's left child, its right child, the symbol it represents, and the frequency it appears with.

2.1 Node Create

To begin with, the first thing we need to be able to do is create a node. To do so, we will allocate space for the node, set the left and right pointers to null, then set the symbol and frequency to whatever was passed in by the function.

2.2 Node Delete

To delete a node, we simply free the passed in node, then set the pointer to null.

2.3 Node Join

To join two nodes, we first sum the frequencies of the two passed in nodes. We then create a new node, with its frequency being the sum, and its character being '$'. We then set the new node's children to the two passed in nodes

2.4 Node compare

We will also need to compare the frequency of two nodes. This function simply returns true if the first node is greater, and false if the left one is.

2.5 Printing a Node

Finally, we need to print a node. To do this, we simply print out its character, its frequency. Alternatively, depending on the function called, we can also simply print out the symbol.

3 Priority Queue

Next, we will need to create a priority queue of nodes, with the nodes with the smallest frequencies having the highest priorities, and being the first to be dequeued. In order to do this, I will be creating a minimum heap, based heavily on my heap sort implementation from assignment 4. As such, contained inside the queue will be an array of pointers to nodes, where the children of a node will be located at the position of the node * 2 + 1 for the left child, and node * 2 + 2 for the right child. The struct for the queue should look like what is shown below.

```
PriorityQueue:
    Node **nodes;
    uint32 capacity
    uint32 current element
```

3.1 Creating a queue

To create a queue, we simply allocate space for the queue, then allocate space for the array of pointers to nodes. Once this is done, we then simply set capacity to the passed in capacity, and current elements to 0.

3.2 Deleting a queue

To delete a queue, we simply free the internal array of nodes, and then the queue itself, before setting the passed in node to null.

3.3 Getting a Node's parent/children

All three of these functions will accept the tested node's position in the heap. To get the left child, we do position * 2 + 1, the right child position * 2 + 2, and the parent (position-1)/2.

3.4 Enqueue

For this function, we are given a priority queue (q) and a node. We start by setting the current element in q to the node. We then save q's current element as position, before incrementing the current element. From here, so long as position is not 0, we compare the frequency of the current node and its parent in the heap. If the parent's frequency is less, we swap the two.

```
bool enqueue(priorityQueue *q, Node *n):
    If capacity equals current element, return false
    uint 32 pos = q's current element
    Increment current element

    q's nodes[pos] = n

    While pos is not equal to 0
        If the node at pos is greater than its parent, swap them

return true
```

3.5 Dequeue

For this function, we are given a queue, and a node pointer pointer that we will use as a return value. We start by setting the return value to the first element in the queue's internal array. Then, we set the current position to 0. After this, we get a value from the bottom of the heap, place it at the top, and move it down the heap by repeatedly swapping it with the smaller of its children. We stop doing this once the node we're testing is smaller than both of its children.

```
bool dequeue(priorityQueue *q, Node **n):
    If current element equals 0, return false

    uint 32 pos = 0

    n = q's nodes[0]

    Decrease q's current element by 1

    position = 0
    create int smaller
    while left child(pos) is less than current element
        if right child(position) is equal to current element
            smaller = right child(pos)
        else
            Use stats to compare left child and right child
            if left child is smaller
                set smaller to left child position
            else
                set smaller to right child position
        if the node at pos is smaller than the smaller child
            break out of the loop
        Swap the node and the smaller child
        Set pos to the position of the smaller child
```

```
return true
```

3.6 Printing a Priority Queue

In order to print the queue, we will start by creating a duplicate of the passed in queue. We start by creating a new, empty queue with queue create, using the arguments from the passed in queue. We then set the current element of the new queue to be the same, including the internal array by iterating over the passed in queue's array, setting the two values to be equal. From there, we print the dequeued node from the new queue so long as dequeue returns true. We then deconstruct the new queue.

```
bool pq_print(priorityQueue *q):
    PriorityQueue queue = pq_create(q's capacity)
    queue's current element = q's current element

    For every value i between 0 and q's current element
        Set queue's nodes[i] to q's nodes[i]

       While dequeue(queue, node) returns true
           node print node

Deconstruct queue
```

3.7 Priority Queue Info Functions

To test if the queue is empty, we simply return true if the current element is 0

Similarly, when testing if it is full, we return true if the current element equals the capacity

Finally, to get the current size of the queue, we simply return the current element.

4 Code

When making our trees, we will need to create codes that correspond to every letter used in a file. Every code will contain an integer that tells us what the current length of the code is, and an array of uint8s which will act as a bit vector. All of this is heavily based on my design for bit vector in assignment 6

4.1 Creating a code

To create the code, all I need to do is create a new code object, set top to 0, then iterate over the internal bit vector, setting every value to 0.

4.2 Getting the size of the code

For this, simply return top.

4.3 Testing if a code is empty

If the top is 0, return true. Otherwise, return false.

4.4 Testing if a code is full

If the top is 256, return true. Otherwise, return false.

4.5 Setting a bit

This function will accept an int i and a pointer to a code, and should then set that bit to 1. Before anything else, we compare i to 256, just to be sure that the i is a valid bit to set. Once this is done, we determine which uint8 the bit is in by doing i/8, saving this as block, before doing i % 8 to determine which bit inside of that uint8 corresponds to i, and saving it as bit. After this, we create a new uint8 j, set it to one, and left shift it by bit position. This should create a number with a 1 at the bit position, allowing us to use it as a bitmask. We then bitwise or the two uint8s to set the bit to 1 if it was not already.

```
bool code_set_bit(Code *c, uint32_t i)

If i is greater than 256, return false

Set block to i/8
Set bit to i%8

Create int j, set it to 1
Left shift j by bit
Bitwise or element block in code's array and j

return true
```

4.6 Getting a bit

We must also be able to get a bit. To do this, we once again start by finding the block and bit in the same way. We then create a duplicate of block at block, before left shifting it over by bit, which should put bit in the first slot. We then test to see if the resulting number is odd. If it is, then we know the requested bit was set, so we return 1. Otherwise, we return 0.

```
bool code_get_bit(Code *v, uint32_t i)

If i is greater than 256, return false

Set block to i/8
Set bit to i%8
```

```
Create int j, set it to the value in element block in c's array
Right shift j by bit

If j is odd
    return true

return false
```

4.7 Clearing a bit

Similarly, we must be able to clear a bit. We start by finding block and bit the same way as above. From there, we must once again go about creating a bit mask, which should consist of a 1 at every position except the bit position. To do this, we first create a new uint8 j with a value of 1, and left shift it by bit-1. From there we xor j with UINT8_MAX, a uint8 with every value set to 1, which should make j be all ones except for the value at 0. We then bitwise and the element at block in the passed-in code, maintaining all values except for the one at bit, which should be set to 0.

```
bool code_clr_bit(Code *c, uint32_t i)

If i is greater than 256, return false

Set block to i/8
Set bit to i%8

Create int j, set it to 1
Left shift j by bit
xor j and UINT8_MAX, and save it in j
Bitwise and element block in c's array and j
```

4.8 Pushing a bit

To push a bit, we start by making sure that there is still room in the code for a bit. From there, if the passed in bit is a 1, we set the bit at the top. If, however, it is a 0, we clear the bit instead. Once this is done, we increment top, then return true.

```
bool code_push_bit(Code *c, uint8_t bit)

If top is greater than 256, return false

If bit is 1, set the bit at top
If bit is 0, clear the bit at top
Increment top
```

```
return true
```

4.9 Popping a bit

To pop a bit, all we need to do is get the bit at the top, subtract 1 from top, then return the fetched bit.

```
bool code_pop_bit(Code *c, uint8_t *bit)

If top is less than 1, return false

Get the bit at top, set it to *bit

Decrease top by 1

return true
```

4.10 Printing a code

Finally, to print a code, all we need to do is iterate over all the bits, get them, then print the result.

```
void code_print(Code *c)

For every value i between 0 and top
    Get bit i
    print bit i
```

5 Input/Output

As part of our encoding, we will, of course, need to read and write bytes to a file. As such, we will create an ADT for doing just that.

5.1 Reading Bytes

The first function will be reading bytes into a buffer, given a file, a buffer, and the number of bytes we'd like to read. To do this, we keep a tally of the number of bytes read so far, and create a variable for the number of bytes read in at one time. We then try to read in the number of bytes we'd like to read - the number of bytes we've read so far until the number of bytes read equals the number of bytes we'd like to read in, or until no bytes are read in at a time, signaling the end of a file.

```
int read_bytes(int infile, uint8 *buf, int nbytes)

int bytes so far = 0
int bytes read = 1

While bytes so far is less than nbytes and bytes read is greater than 0
    Try to read nbytes-bytes so far in from infile to buffer + bytes so far with
    read()
    Add the return value of above to bytes read
    Add bytes read to bytes so far

return bytes so far
```

5.2 Writing Bytes

Writing bytes is almost identical to reading them. We follow exactly the same formula, except now we try to write bytes, instead of reading them.

```
int write_bytes(int outfile, uint8 *buf, int nbytes)

int bytes so far = 0
int bytes wrote = 1

While bytes so far is less than nbytes and bytes wrote is not 0
    Try to write nbytes to outfile from buffer + bytes so far with write()
    Add the return value of above to bytes read
    Add bytes read to bytes so far

return bytes so far
```

5.3 Reading individual bits

We would also like to read individual bits from a file. This is, however, impossible to do directly, so we will instead create a static variable that holds a number of bytes of text, another that will track what the current index is for that block, and a final third static variable that tracks the number bytes that were actually read into the block. Using these, if the current index equals the number of bytes read in, we use the above read_bytes function to try and read more bytes. If the number of read bytes is equal to 0, we return false, as we have read the end of the file. Otherwise, we get the bit at the current index, add 1 to the current index, and return the bit and true.

```
static uint8* block
```

```
int current index
int read in bytes

bool read_bit(int infile, uint8 *bit)

If read in bytes = 0, return false

if current index is greater than read in bytes * sizeof uint8
    read in bytes = read_bytes(infile, block, BLOCK)
    set current index to 0

If read in bytes = 0, return false

Set element to current index/8
Set a to current index%8

Create int j, set it to the value in element element in block
Right shift j by a

If j is odd
    set *bit to 1
else
    set *bit to 0

return true
```

5.4 Write Codes

Here, we would like to store individual bits. As such, we will do much the same thing as when we were reading individual bits. Using a static block, and an int representing the index, we write every bit of the inputted code to the block. Once the buffer is full, we simply write the whole buffer to the outfile, and start writing again from index 0.

```
static uint8* block
int current index

void write_code(int outfile, Code c)

For every value between 0 and the size of c
    If index / 8 = the size of block
        Write the size of block bytes from block to the outfile with write bytes
    Set element to current index/8
    Set a to current index%8
    Left shift 1 over by a, save it as j
    Or block[element] with j
    Increment index by 1
```

5.5 Flushing the codes

Finally, we may reach the last code, and still not have taken up all the room in the buffer. When this happens, we use flush codes to write all of the codes currently contained in the buffer to the outfile. We do this by simply writing the the index bits from the block to the outfile

```
static uint8* block
int current index

void flush_codes(int outfile)
    Set element to current index/8
    Set a to current index%8
    Set j to UINT8_MAX
    right shift j by 8 - a
    and j and block[element]

    Write index/8 + 1 bytes to outfile with write bytes
```

6 Stack

After this, we will need to create a stack. This stack will simply hold a number of nodes, with traditional last in first out stack behavior. Every stack will contain a top index, a capacity, and an array of nodes.

6.1 Creating a stack

Given a capacity, we simply allocate space for the stack, set the internal capacity to the passed in capacity, use calloc to allocate space for capacity node pointers, and set the top to 0.

6.2 Stack Delete

To delete a stack, we simply free the internal array and then the stack itself, before returning null.

6.3 Stack empty

If the top is 0, return true. Otherwise, return false.

6.4 Stack full

If the top equals the capacity, return true. Otherwise, return false.

6.5 Stack Size

Return top

6.6 Stack Push

To push a node, all we need to do is set the element at the index of top to the passed in node, before incrementing top. We return false if top equals capacity, and true if it does not.

6.7 Stack pop

To pop the stack, all we need to is reduce top by 1, then return the element at the index of top. If the stack is empty, we return false, otherwise, we return true.

6.8 Printing a stack

To print a stack, we simply iterate over all the nodes contained in it, and print them as we go.

7 Huffman

Finally, using everything created above, we will need to be able to create the actual huffman tree.

7.1 Building a Tree

To build our tree, we are given an array of ints, where the index of the array corresponds to the character in question, and the value contained in that index is the number of times it appeared. Ie, index 65 having a value of 32 means that As appeared 32 times throughout the file. We then iterate over all the nodes, creating a node for each if the frequency is not 0, and entering them into a priority queue. Then, while the queue contains at least two nodes, we pop the queue twice, join the resulting two nodes, and add the new joined node back into the queue. Once there is only one node left in the queue, this is the root of the tree, which we then return.

```
Node *build_tree(uint64_t hist[static ALPHABET])
    Create a new priority queue with a capacity of ALPHABET

    For every value i between 0 and ALPHABET
        If hist[i] is not 0
            Create a new node with i and hist[i], and enqueue it

    While the queue’s size is greater than 1
        Pop the queue once to get the left node
        Pop the queue a second time to get the right node
        Use node join to enjoin these two nodes
        Add the resulting node back to the queue

Pop the queue one last time, and return the result
```

7.2 Building the Codes

To build our set of codes, we are passed in an array of codes and the root of a huffman tree. We start by initializing a previously declared static code, then do a post order traversal of the tree. This means that if the node is not a leaf, we push a 0 to the code, then call build codes on the left child. We then pop a bit, push a 1 to the code, then call build codes on the right child. Finally, we pop a bit before the function ends. If the node is a leaf, however, we save a copy of code into the corresponding position in the code array.

```
static current code

static void code traversal(Node *current, Code table[])
    If current has no children
        table[node's symbol] = current code
        return
    Push a 0 to current code
    code traversal(node's left, table)
    Pop current code
    Push a 1 to current code
    code traversal(node's right, table)
    Pop current code

void build codes(Node *root, Code table[])

    code = code init
    code traversal(root, table)
```

7.3 Dumping a Tree

To dump a tree, all we need do is conduct a post order transversal of the tree. If the node is a parent, we call the dump tree function on the left child, then the right child, before writing an I to the outfile. If, however, it is a leaf, we instead write an L to the outfile, followed by the node's internal character.

```
void dump tree(int outfile, Node *root)
    If node has no children
        Write an L to the outfile
        Write root's character to the outfile
        return

    Call dump tree on root's left child
    Call dump tree on root's right child
    Write an I to the outfile
```

7.4 Rebuilding a Tree from the Dump

When rebuilding a tree from a file post dump, we are given the number of bytes it is contained in, and an array containing the read in dump. We start by creating a stack and iterating over the entire dump array. If the next character is an L, we create a new node with the next character the array, and push that to the stack. If the next character is an I, we pop the stack once to get the right child, and pop it again to get the left child (THIS ORDER IS THE OPPOSITE OF WHAT IT IS EVERYWHERE ELSE! DON'T FORGET THAT!). We then join these two nodes, and add the new parent node to the stack. Once the stack only contains a single node, that will be the root of our tree.

```
void rebuild tree(int nbytes, int tree dump)
    Create a stack

    For every value i between 0 and nbytes
        If tree dump[i] is an L
            Create a new node containing tree dump[i+1]
            Push the new node to the stack
            Add 1 to i
        If tree dump[i] is an I
            Pop the stack once to get the RIGHT node
            Pop the stack a second time to get the LEFT node
            Enjoin the two nodes, and add the new parent to the stack

     Pop the stack one last time, and return the result
```

7.5 Deleting a Tree

To delete a tree, we simply use the same post order traversal we have been using before, deleting the nodes as we go.

8 Encoder

Next, we need to create a program that will generate a huffman tree based on a file. To do this, we start by processing command line inputs, before creating an array that will represent the frequency of all of the ascii characters. We then iterate over all the characters in the infile, incrementing the corresponding value in the histogram. Using this histogram, we run build tree as it is defined above to construct our tree. Then, using the tree, we run build codes to construct the code for every ascii character. From here, we write the provided magic number, the permissions that the input file had (found with fstat), the size of the huffman tree dump, and finally, the size of the file before compression to the outfile. We then write the header to the outfile, alongside the tree with dump tree. Finally, we iterate over the infile again, using write code defined above to write the code for each character to the outfile.

```
Set defaults, then handle command line inputs

Open the infile
Open the outfile
Create a header struct

int unique characters

Get the permissions for infile with fstat, save it in header
Set the outfile permissions to the infile permissions

Create an array in that is block long

Create an array arr that is 256 items long

Read characters to arr from the infile
Allocate the that many more slots in in
While read in bytes is not 0
    For every byte that was read in
        Increment arr[current char]
        if arr[current char] is 1, increment unique characters as well
        increment counter

If arr[0] is 0, increment it.
If arr[1] is 0, increment it as well.

Node *tree = build tree with arr

Create another array codes that is 256 characters long, and set it to build codes
with tree

Save the header's magic number to the provided magic number, and set header's tree
size to 3 * unique symbols - 1
Get the length of the infile with fstat, and set the header's length

Write size of header bytes to the outfile with write bytes

Dump tree with dump tree

Iterate over all the characters in the file again
    Use write code to write code[current char]

Flush codes
```

```
Close the files.
```

9 Decoder

Finally, given the file made in encoder, we have to be able to undo it. To do this, we will create a decoding function. Same as the last time, we start by processing command line inputs. We then try to read in a header struct from the infile. If the header is not read successfully, or if the magic number is wrong, we print an error message and return 1. From here, we then set the permissions of the outfile to be the same as the header's permissions with fchmod. Next, we use rebuild tree to reconstruct the huffman tree that was dumped after the header in encode. From here, we begin to read in individual bits from infile, counting every bit we read in. At every bit, we take the path down the huffman tree specified by that bit, i.e. a 1 means go right, and a 0 means go left. Once we reach a leaf, we write the symbol that corresponds to that leaf to a buffer, writing the buffer to the outfile whenever it is full, and then return back to the root. We end by writing whatever is left in the buffer and closing the files.

```
Set defaults, then handle command line inputs

Open the infile
Open the outfile
Create read in a header struct from the infile

If header's magic number is not equal to the provided magic number, or the header
failed to be read in
    Print an error message, and return 1

int unique characters

Get the permissions for the outfile from header, set them with chmod

Create an array that is header's tree size bytes long

Rebuild the tree with rebuild tree, set that to root

Current node is equal to root
Create a counter, set it to 0
Create an array arr that is block elements long

While counter is not equal to header's file size
    Read in the next bit from the infile
    If that bit is 0,
        current node = current node's left child
    Otherwise
        current node = current node's right child
    If current node has no children
        Write the symbol in the current node to arr[counter % block]
        Increment counter
        If counter % block equals 0
            Write block characters to the outfile

Write counter % block characters to the outfile

Close the files
```