DESIGN: Assignment 5 – Huffman Coding

Nelly Sin

October 2021

1 Functionality

The user will input a file to encode by using the Huffman coding and it will use a decoder to decode the compressed file. This can be confusing at first, but the functions will all break into an ADT– reading the file, using the Huffman tree to traverse through the input. Encoding the symbols from an infile and decoding the symbols to an outfile.

2 The Encoder

First, we use a Huffman encoder. After reading the input of a file, the Huffman encoding will compress the file.

2.1 Command-Line

The user must type the follow:

1. -h: Printing out the menu and expanding the message that describes how the program works and what inputs this program takes in.
2. -i infile: This specifies the input file the Huffman code will encode (defaulting to stdin).
3. -o outfile: specifies what file to write after compressing the input defaulitng to stdout).
4. -v: printing the compression statistics to stderr. This includes the uncompressed file size, compressed file size, and the space saving. This is the formula for calculating space saving:
   100 x (1-(compressed size/uncompressed size))

2.2 Priority Queues

This is the first ADT I will be implementing. I will be using the Heap sort implementation from assignment 3 to serve as a reference when using the min heap in the priority queue.

If the left if less than the capacity and the current frequency is greater than the left node. The smallest node (left) will be the new smallest, and vice versa with the right node. But if they are not
When the smallest frequency is not the parent. Then we have to swap the parent with the current node's frequency.

Similar to assignment 4, the function in "PriorityQueue" will be the constructor and the priority queue's maximum capacity is specified by capacity.

"pqdelete" is a function for freeing the priority queue and making the pointer set to NULL (for no memory leaks).

"pqempty" is a function that returns true if the priority queue is empty, but if else, return false.

If the priority queue pointer is at 0, then this will return true (indicating that the priority queue is empty).

"pqfull" is a function that returns true if the priority queue is full and if not it will return false.

If the priority queue pointer is at the capacity, then return true (indicating the priority queue is full).

"pqsize" is a function that returns the number of items that are currently in the priority queue.

Returns where the pointer is in the priority queue.
"enqueue" is a node into the priority queue. So, if the enqueue is full before enqueuing the node then return false. If not, return true.

"dequeue" is a node in the priority queue. The node that is being dequeued must have the highest priority over all other nodes in the priority queue. This returns back to the min heap sort, how top of the tree has the highest priority. However, if the priority queue is empty prior to dequeuing a node return false, otherwise return true.

This checks if the priority queue is full (return false) else if it's empty then the element on the current pointer is the current node. We will have to initialize the parent to the current pointer and fixing our heap (provided in assignment

3). We will be fixing the heap in enqueue to sort the parent nodes.)

"pqprint" This function is not necessary for the final submission, however it's useful for debugging the priority queue. Printing will display a tree – provided by TA Eugene.

3 Codes

This means maintaining the stack of bits while backtracking in the tree to create a code for each individual symbol. This breaks into three different scenarios, setting a bit, clearing a bit, and getting the bit.
BLOCK: will be the block of symbols to read from ALPHABET 256: The maximum size of in ASCII.
MAGIC: Knowing if this is a vaid file.
MAXCODESIZE: will be the maximum number of bytes needed to store any valid code.
MAXTREESIZE(3 * ALPHABET - 1): the maximum Huffman tree dump size.

"codeinit" Is initializing the constraints of the Huffman algorithm. Creating the new code on the stack and setting the top to 0 and zeroing out the array of bits.

"codesize" Will return the size of the code – also the number of bits that is being pushed on to the stack.

Initialize code and set everything to 0.

"codeempty" will tell us by a boolean if the Code is empty – true or not – false.

This will return the top index of the code

"codefull" Will tell us by a boolean if the Code is full – true or not – false.

Indicating if the top index is 0 then it is empty (true).

"codesetbit" Setting an index i in code and setting it to 1. However, if it is out of range, return false, else return true (provided by Professor Long).

Translate the byte to a bit and shift it by i to set the bit to 1

"codeclrbit" Clearing the index i in Code – will be turned to 0. If i is out of bounds then it will return false, otherwise return true (provided by Professor

Long).

Translate the byte to a bit and shift it by i to set the bit to 0

"codegetbit" Retrieve the bit at index i in the Code. If i in out of bounds, or if i is 0 then return false. Otherwise, return true only if the i is 1 (provided by Professor Long).

Translate the byte to a bit and it will return the bit of the translated byte

"codepushbit" Pushing the bit into the code. This function will return false if code is full and true otherwise.

```
    if (top == capacity)
        return false
    else:
    if (bit == 1)
        set the bit
increment the top
    else
        clear the bit
increment the top
```

"codepopbit" Popping off the code by passing it off to the pointer "bit". If if was successfully popped, meaning that the stack did not reach to 0, then return true and false otherwise.

```
    if (top == 0)
        return false
    else:
        decrement the top
        get the bit
        clear the bit
        increment the top
        return true
```

"code print" This will be a simple for loop that loops through the code

4 I/O

"readbytes" we will be reading bytes in the text file. 'nbytes', one of the function's parameter will tell us how many bytes to read in the file. This function will be a while loop that conditions whether or not the number of bytes read has reached 0.

```
while bytes > 0:
    bytes will equal to reading the infile
    keep track of how many bytes are read in total
    return the total read
```

"writebytes" writing bytes is very similar to reading bytes. It will write out the bytes to the outfile. Again, 'nbytes' will tell us how many bytes to write to the outfile.

```
while bytes > 0:
    bytes will equal to how many will write to the outfile
    keep track of how many bytes are written in total
    return the total written
```

"readbit" will be calling read bytes to read each block of bytes in the buffer. This will return false when there are no more bits that can be read and true when there are still bits to read.

```
if byte is empty:
    bytes will equal to the how many bytes are read from the infile
    reading the last valid byte
returning the bit out of the buffer by getting the bit and increment the index of the buffer
```

"writecode" from reading the bit we must store the bits to Code c, the buffering into the buffer. And once the buffer of BLOCK is filled write out the contents of the buffer.

```
if code if less than code size
    get bit
    if bit equals 1
        set the bit
    else
        clear the bit
returning the bit out of the buffer by getting the bit and increment the index of the buffer
```

"flushcode" make sure you have written and read the given bytes, thus there is no data leaks.

```
if index module 8 equals to 0
    it will equal to index / 8
else
    index / 8 + 1
call writebytes
```

5 Stacks

"stackcreate" we will first allocate memory for stack and initializing capacity and top to 0. The stack "s" will be a pointer to items.

"stackdelete": this is to delete stack once we are finished using stack. It will free the memory allocated by the stack.

"stackempty": this is a boolean function and will return true when the stack is empty and false otherwise.

```
if (top == 0)
    return true
else
    return false
```

"stackfull": this is a boolean function and will return true when the stack is full and false otherwise.

```
if (top == capacity)
    return true
else
    return false
```

"stacksize": this returns the number of nodes in the stack.

return the top of the stack

"stackpush": this will push a node to the stack. This is a boolean function because it will return true if pushing the node of stack is successful and false otherwise. It is false when the stack is full.

```
if stack full == true
    return false
else
    push node to the top of the stack
    increment the top
```

"stackpop": this will pop the node off the stack. This is a boolean function so it will return true if popping the stack is successful and false otherwise. It returns false when the stack is empty.

```
if stack empty == true
    return false
else
    pop node from the top of the stack
    decrements the top
```

6 Huffman Coding

"buildtree": this function will return the root of the tree or the root node that is constructed tree.

```
We will be min heaping every parent node –initialize the parent
while the size of pq is greater than 1
    dequeue the left
    dequeue the right
    enqueue the nodejoin

dequeue the root from pq
delete the pq after (no memory leaks)
returning the root
```

"buildcode": after constructing codes are copied to the code table, 'table', that has the ALPHABET indices, each index is for each symbol.

```
This is traversing     if the left and right nodes are not leaf nodes
    then the code will be the symbol
else
    if the left is interior
        push the bit
        build the code with the left node
        pop the bit – go back up
    if the right is interior
        push the bit
        build the code with the right node
        pop the bit – go back up
```

"dumptree": using the post-order traversal for the Huffman tree from the 'root' which writes it to the outfile. 'L' will be followed by the byte of the symbol for each of the leaf and 'I' for the interior nodes. Interior node does not have a symbol.

```
if the root is interior
    do post order traversal to right (call the function itself)
    o post order traversal to left (call the function itself)
if the left and right root is a leaf node
        call write bytes for left or right
        write the bytes for the interior nodes
```

"rebuildtree": this is the reconstructing of the Huffman tree because after tree dump, this is where it will be stored in the array. It returns the root node of the reconstructed tree.

```
create a stack
    if we know the element of the tree index is L
    this is a leaf node thus the next would be a symbol
if we know the element of the tree index is I
    we know the left and right nodes so we pop them
    do node join for left and right and push the parent to the stack
        write the bytes for the interior nodes
we will be returning the root, so we would be popping and deleting the stack
```

"deletetree": after finishing the rebuild tree and using the tree it will delete it and freeing all allocated memory – for no memory leaks.

```
if root exists
    if the left is an interior
    traverse back
    if the right is an interior traverse back
    traverse back
we will be deleting the root
```

7 Encode

In the beginning, I do not understand the encoding portion, because there were a lot going on in each ADT. But later, I was able to understand how the huffman, priority queue, stacks, etc. that came together and how each links to one another to encode.

We will be encoding a text file – this can be a large or small file. Encode will be linking to the Huffman tree.These some vague steps for creating the encode.

The encode file will be one of the main files we will have to create with command lines of taking in the infile, verbosing the statistics of the compression of the file, and outputting to the outfile.

I think encode will be very difficult, considering that we have to implement the ADTs we have covered so far. But, debugging will be hectic as we have to look through every function, in each file. Making sure that they work properly. This means I must test every file to see the the ADTs are all functional and passing.

Here are some of the steps I will be taking. These are not the actual steps I may take, but it's a vague understanding of how I understand it.

1. We will first create the histogram.
2. Then construct the header, this is to be set to be a "magic" number (0xBEEFD00D).

3. We will have to call the Huffman tree with priority queue, buildtree.
4. Build codes, this is where we will be encoding with the symbols, leaf nodes, and interior nodes by traversing the huffman tree.
5. Don't forget to dump the tree and close the infiles and outfiles!
6. We will also have to include all the statistics for the verbose.This includes the pre-compressed size, compressed size, and space of the file.

8 Decode

I also did not quite understand the decoding portion fully, however, as soon as I continued reading the assignment doc, I have a pretty clear understanding of what I am suppose to implement.

After encoding, we will be decoding the encoded text file. The decode file is our second main file that will have the infile, verbosing the statistics of the decompression of the file, and outputting the decoded result to a outfile.

I think The decode implementation is pretty straight forward considering it takes in code and reverse back to the text's original state. And looking back to the assignment doc. I understand the implementation, of rebuilding the tree in order to traverse back.

1. We will have to construct a histogram with an array of 256.
2. The increment the count of element 0 and element 255 by one in the histogram. They will have two elements the same.
3. Construct the Huffman tree using the priority queue, this is also one of the ADT's we have implemented.
4. Constructing a code table to traverse the Huffman tree. Using one of the ADTs we have built called buildcodes. With an array of 256.
5. Then we must construct the header, with the given header file and it's permissions.
6. Write out the outfile using dumptree, one of the ADTs we have implemented.
7. Next we must write the code using writecode() and flushcodes() to flush out the remaining buffered codes.
8. Finally, remember to close the infile, outfile, and free any remaining memory for no memory leaks!

9 CITATION

Thank you to the TA's, tutors, and Professor that provided a lot of help and clarity to the assignments.

This includes any pseudo-code and any additional (further) help and guidance in this assignment is credited to them.

CITE: TA Eugene, Tutor Ben, Tutor Miles, Tutor Brian, Tutor Jason, Tutor Eric, and Professor Long. (All credit goes to their help and given pseudo code throughout this assignment.)