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, and decoding the symbols.

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.

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.

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

”pqsize” is a function that returns the number of items that are currently 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.

”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.

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

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

"codesetbit" setting an index i in code and setting it to 1. However, if it is out of range, return false, else return true.

"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.

"codegetbit" retrieves 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.

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

"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.

"code print" I currently do not know much about codeprint as it's a debugging function.

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.

"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.

"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.

"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.

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

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.

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

"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.

"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.

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