Assignment Six Design.pdf

By James Yim

Description of Program:

This program will be able to encode and decode text to a smaller representation. It will be able to take in input from a file and encode the contents compressing the file. It will also be able to decode a compressed file, outputting the original file.

Files to be included in asgn6

- encode.c:
    - This file will contain the implementation of the Huffman encoder.
- decode.c:
    - This file will contain the implementation of the Huffman decoder.
- defines.h:
    - This file will contain the macro definitions used throughout the assignment.
- header.h:
    - This will contain the struct definition for a file header.
- node.h:
    - This file will contain the node ADT interface.
- node.c:
    - This file will contain the implementation of the node ADT.

- pq.h:
    - This file will contain the priority queue ADT interface. This file was provided by the professor.
- pq.c:
    - This file will contain the implementation of the priority queue ADT.
- code.h:
    - This file will contain the code ADT interface. This file was provided by the professor.
- code.c:
    - This file will contain the implementation of the code ADT.
- io.h:
    - This file will contain the I/O module interface. This file was provided by the professor.
- io.c:
    - This file will contain the implementation of the I/O module.
- stack.h:
    - This file will contain the stack ADT interface. This file was provided by the professor.
- stack.c:
    - This file will contain the implementation of the stack ADT.
- huffman.h:
    - This file will contain the Huffman coding module interface.

- huffman.c:
    - This file will contain the implementation of the Huffman coding module interface.
- Makefile
    - This will help compile and run the program with the necessary files.
- README.md
    - This describes how to use the program. It will explain command line options that the program accepts. It will also note bugs and errors that are in the program.
- DESIGN.pdf
    - This pdf will show the process of how to create this program. It will include pseudocode and it will credit sources where information was learned.

Design and Pseudocode

Nodes

Creating Nodes

```
Node *node_create(symbol, frequency)
{
        allocate memory for the node
        if (successfully allocated)
        {
                set the node's symbol to the symbol input
                set the node's frequency to the frequency input
                set its left and right nodes to NULL
        }
        return node
}
```

Deleting Nodes

```
void node_delete(node)
{
	free the symbol and the frequency
	free the actual node container
}
```

Joining Nodes

```
Node *node_join(left node, right node)
{
	add the frequencies of the two nodes
	create a parent node using the combined frequencies
	set the parent node's left and right to the corresponding nodes
	return the parent node
}
```

Priority Queues

Creating the queue

```
PriorityQueue *pq_create(capacity)
{
	allocate space for the priority queue
	if (allocation was successful)
	{
		set queue capacity to input capacity
		set tail and head of queue to 0
		allocate space for nodes
		if (allocation was successful)
		{
			return the queue
		}
		else
		{
			free the queue
		}
	}
return NULL // this is if the queue was unsuccessfully created
}
```

Deleting the queue

```
void pq_delete(queue)
{
	if (the queue isnt empty)
	{
		free the nodes
		free the queue
	}
}
```

Checking if the queue is empty

```
bool pq_empty(queue)
{
	if (queue head and queue tail are equal)
	{
		return true;
	}
	return false;
}
```

Checking if queue is full

```
bool pq_full(queue)
{
	if (queue exists)
	{
		if (queue head + 1 equals the tail)
		{
			return true
		}
	}
	return false
}
```

Getting queue size

```
uint32_t pq_size(queue)
{
	return the queue head
}
```

Enqueuing an item

```
enqueue(PriorityQueue, item)
{
	if (PriorityQueue exists)
	{
		if (check if the queue is full)
		{
			return false
		}
		put the item at the head of the queue
		sort the queue so that the the item with the highest priority is at the tail.
		loop through the queue and compare the queued item with each element.
		place item in its respective spot
		return true;
	}
	if (priority queue doesnt exists)
	{
		return false;
	}
}
```

Dequeuing an item

```
dequeue(PriorityQueue, outputItem)
{
	if (queue exists)
	{
		if (queue is empty)
		{
			return false;
		}
		put  the node at the tail of the queue into the outputItem
		shift all the items in the priority queue down to get a new item at the tail
		return true;
	}
	if (queue doesnt exist)
	{
		return false;
	}
}
```

Codes

Initializing the code type

```
Code code_init()
{
	declare a code variable
	declare the top of the code setting it to 0
	return the code variable
}
```

Checking code size

```
code_size(code)
{
	return the top of the code
}
```

Checking if code is empty

```
code_empty(code)
{
	if (the top of the code == 0)
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

Checking if code is full

```
code_full(code)
{
	if (the top of the code == MAX_CODE_SIZE)
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

setting a bit in the code

```
code_set_bit(code, index)
{
	if (the index is greater than the size of the alphabet or if it equals 0)
	{
		return false;
	}
	perform a bitwise function to set the bit at the index to 1
	return true;
}
```

clearing a bit in the code

```
code_clr_bit(code, index)
{
	if (the index is greater than the size of the alphabet or if it equals 0)
	{
		return false;
	}
	perform a bitwise function to set the bit at the index to 0
	return true;
}
```

getting a bit in the code

```
code_get_bit(code, index)
{
	if (when we perform a bitwise operation at the index and it returns 1)
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

pushing a bit

```
code_push_bit(code, bit)
{
	if (code is full)
	{
		return false;
	}
	set the bit at the top of the code to the input bit
	increment the top by one
	return true;
}
```

popping a bit

```
code_pop_bit(code, bit)
{
	if (code is empty)
	{
		return false;
	}
	place the bit at the top of the code into the output bit
	decrement the top by one
	return true;
}
```

Input and Output

Reading Bytes

```
read_bytes(infile, buffer, numOfBytes)
{
	make a counter to track total bytes read *read_count*
	make a counter to track bytes read as that moment *current_read_bytes*

	do
	{
		set *current_read_bytes* = to the bytes read from read(infile, buffer +
	read_count, numOfBytes - read_count);
		add the current_read_bytes to the read_count
	}
	while (we havent read all the required bytes OR the file has bytes to read still)
	return read_count;
}
```

Writing Bytes

```
write_bytes(infile, buffer, numOfBytes)
{
	make a counter to track total bytes written *write_count*
	make a counter to track bytes written as that moment *current_write_bytes*

	do
	{
		set *current_write_bytes* = to the bytes written from rwrite(infile, buffer +
	write_count, numOfBytes - write_count);
		add the current_write_bytes to the write_count
	}
	while (we havent written all the required bytes OR the bufferhas bytes to wrute still)
	return write_count;
}
```

Reading a bit

```
read_bit(infile, outputBit)
{
	if (the buffer is empty)
	{
		set the bit index to 0
		read_bytes from the infile
	}

	current_byte equals the buffer[bit index / 8]
	bit = current_byte right shifted by the bit index / 8
	AND bit with 1
	put bit into the outputBit

	increment the bit index
	if (we read the whole buffer)
	{
		return false;
	}
	else
	{
		return true;
	}
}
```

Writing the Code

```
write_code(outfile, code)
{
	for (bit index < size of code)
	{
		if (the bit at bit index is 1)
		{
			set that bit in the buffer to 1 using bit vectors
		}
		if (the bit at bit index is 0)
		{
			set that bit in the buffer to 0 using bit vectors
		}
		if (buffer is full)
		{
			write buffer to outfile
		}
	}
}
```

Flushing codes

```
flush_codes(outfile)
{

	while (we are not at the end of the last byte)
	{
		set uninitialized bits to 0 using bit vectoring
	}
	write out the buffer to outfile
}
```

Huffman Coding

Building a Huffman Tree

```
build_tree(histogram of symbols)
{
        create a priority queue with the histogram of symbols
        for (symbols with a frequency greater than 0)
        {
                create a node with the symbol and the frequency
                queue the created node
        }

        while (there are nodes in the queue)
        {
                dequeue the first node making it the left node
                dequeue the 2nd node making it the right node
                make a parent out of the two nodes
                queue the parent back into the queue
        }

        return the last node in the queue as the root
}
```

Building a code table

```
build_code(node, code table)
{
	if (a root node exists)
	{
		if (the node is a leaf)
		{
			put node into the table
		}
	}
	for left bits
	{
		push a 0 into the code
		call build_code for the left node
		pop the last bit in the code
	}
	for right bits
	{
		push a 1 into the code
		call build_code for the right node
		pop the last bit in the code
	}
}
```

Dumping the tree\

```
 dump_tree(outfile, root node)
{
	if (root node exists)
	{
		dump the left node
		dump the right node
		if (no left or no right)
		{
			write an L into the outfile followed by the symbol of the node
		}
		else
		{
			write an I into the outfile
		}
	}
}
```

Rebuilding the tree\

```
rebuild_tree(nbytes, tree)
{
        make a stack
        while there are still nodes in the tree
        {
                if we encounter an L
                {
                        the next node is a symbol and we push it onto the stack
                }
                if we encounter an I
                {
                        the next node is the left child
                        and the second node is the right child
                        combine them and make a parent node
                        push the parent node onto the stack
                }
        }
        return the root of the tree
}
```

Notes about code

- Encode and Decode were not finished in this version.
- We use bit vectoring in order to change the values of individual bits because we can only manipulate bytes as the smallest form of information.

Error Handling

- Encode and Decode were not finished in this version

Credit

- Eugene's sections helped with how to get an idea with how to start the project
- Brian's section also helped with how to visualize the buffer and how to write into it
- The CSE 13s discord helped with troubleshooting errors I had.