Shaan Mistry
20 February 2022
Professor Long

Assignment 6: DESIGN.pdf

Description of Program:
This program using Huffman Coding to compress and decompress files.

Huffman Coding:

- Give each symbol a unique bit standing (code).
- Frequently appearing symbols are given shorter codes while infrequently appearing symbols are given longer codes.

In order to assign each symbol with a unique code, the program creates a Huffman Tree. In order to make the Tree, the program first creates a histogram of the frequency of each unique symbol.

Huffman Tree:

- Each symbol is represented as a node with a symbol and a frequency.
- We create a priory queue for these nodes with the highest priority being the node with the smallest frequency.

Files to be include in "asgn6":

1. encode.c
   - Implementation of the Huffman encoder.
   - Parses command line options.
   - Inputs a file to be encoded and compressed and stores the compression in the output file.
2. decode.c
   - Implementation of the Huffman decoder.
   - Parses command line options.
   - Inputs a compressed file and decompresses it and stores the results in the output file.
3. defines.h
   - Contains macro definitions.
4. header.h
   - Contains struct definition for a file header.
5. node.h
   - Interface for the node ADT.
6. node.c
   - Implementation of the node ADT.
   - Allows the creation of nodes with a symbol and a frequency.
7. pq.h
   - Interface for the priority queue ADT.
   - Uses min heap to sort to prioritize nodes.
8. pq.c
   - Implementation of the priority queue ADT.

9. code.h
   - Interface for the code ADT.
10. code.c
    - Implementation of the code ADT.
11. io.h
    - Contains the Input/Output module interface.
12. io.c
    - Implementation of the Input/Output module.
13. stack.h
    - Interface for the stack ADT.
14. stack.c
    - Implementation of the stack ADT.
15. Huffman.h
    - Interface for the Huffman coding module.
16. Huffman.c
    - Implementation of the Huffman coding module.
17. Makefile
    - Directs compilation processes for all **C** files.
    - Formats .c and .h files.
    - Cleans object files and executables.
18. README.md
    - Markdown Format.
    - Contains information on how to build the program, how to run the program, and how to clean up and object files and binaries created from building the program. It can also contain information on possible bugs or errors in the program.
19. DESIGN.pdf
    - This file.
    - Contains pseudocode and explanations of the program.

Pseudocode/ Structure:

<u>encode.c</u>

```
WHILE getopt finds command-line arguments:
        SWITCH getopt:
                Case h: print out help message.
                Case i: set input file to getopt argument.
                Case o: set output to getopt argument.
                Case v: set statistics to true.

Create an array of 256 unsigned 64-bit integers which represent the histogram.
Increment the count of element 0 and element 255 by 1.

WHILE the end of file has not been reached:
        Read symbol in the file.
        Set i to the ascii value of the symbol.
        Increment the ith element of the histogram by 1.

Construct a Huffman tree using the function build_tree().


Construct a code table using build_codes().
Construct a header struct.
        Set magic number to
        Set permissions to the permission bits of the input file.
        Set the tree_size to (3 x unique symbols) – 1.
        Set the file_size to the size in bytes of the input file.

IF statistics is True:
        Print: Uncompressed file size, Compressed file size, and space saving: 100 x (1 -
(compressed size / uncompressed size)) to stderr.

Print the constructed header to the specified output file.
Write the constructed Huffman tree to the output file using dump_tree().
FOR each symbol in the input file:
        Wrote the symbol’s code to the output file using write_code().
Flush remaining buffered codes using flush_codes().
Close the input and output files.
```

decode.c

```
WHILE getopt finds command-line arguments:
        SWITCH getopt:
                Case h: print out help message.
                Case i: set input file to getopt argument.
                Case o: set output to getopt argument.
                Case v: set statistics to true.

Read in the header from the input file.
IF magic number does not equal 0xBEEFBBAD:
        Return an error.
Set the permissions of the output file to the read permission in the header.
Read the dumped tree from the input file into an array of size tree_size bytes.
Reconstruct the Huffman tree using the function rebuild_tree().

Read the infile one bit at a time using read_bit().

// Walk the Tree: Pesuedocode inspired from Eugene’s Lab Section 2/18/2022.

Close the input and output files.
Read the emitted tree from the input file.
```

```
IF statistics is True:
    Print: Uncompressed file size, Compressed file size, and space saving: 100 x (1 - (compressed size / decompressed size)) to stderr.
```

node.c:

```
FUNCTION node_create:
    Allocate memory for a node struct.
    Set the symbol of the node to the inputted symbol.
    Set the frequency of the node to the inputted frequency.
    Return a pointer to the node.

FUNCTON node_delete:
    Free the memory allocated to the node struct.
    Dereference the pointer pointing to the node struct.

FUNCTION node_join:
    Create a new node with the symbol “$” and a frequency of the sum of the left and right node’s frequencies.
    Set the left child node of the node to the inputted left node.
    Set the right child node of the node to the inputted right node.
    Return a pointer to the node.

FUNCTION node_print:
    Print the symbol and frequency of the inputted node.
```

pq.c:

```
Define the Priority Queue struct with head, tail, capacity, and an array of nodes.

FUNCTION pq_create:
    Allocate memory for a Priority Queue struct.
    IF the memory allocation for the Priority Queue succeeded:
        Set the queue head and tail to 0.
        Set the queue capacity to the inputted capacity.
        Allocate memory for the array of nodes with a size of the inputted capacity.
        IF the memory allocation for the array of nodes succeeded:
            Return a pointer to the priority queue struct.
        ELSE:
            Deallocate the memory for the Priority Queue struct.
    Return a Null pointer.

FUNCTION pq_delete:
    IF the pointer to the Priority Queue is not null:
        Deallocate memory for the array of nodes.
        Deallocate the memory for the Priority Queue struct.
        Deference the pointer to the Priority Queue.

FUNCTION pq_empty:
    IF the priority queue tail is 0.
        Return True.
    Return False.
FUNCTION pq_full:
    IF the index of the priority queue tail is equal to the capacity:
        Return True.
    Return False.

FUNCTION pq_size:
    Return the tail of the priority queue.

FUNCTION enqueue:
    IF the priority queue is full:
        Return False.
    Set the tail index of the node array to the inputted node.
    Fix the heap.
    Increment tail by 1.
    Return True.

FUNCTION dequeue:
    IF the priority queue is empty:
        Return False.
    Set the inputted pointer to the 0th index of node array.
    FOR i in range (0, priotiy queue capacity):
        Set the ith element of the node array to the i+1 elemtn of the node array.
    Decrement tail by 1.
    Fix the heap
    Return true.


FUNCTION: pq_print:
    FOR each node in the priority queue:
        Print symbol and frequency.
```

code.c

```
FUNCTION code_init:
    Create a code struct.
    Set top to 0.
    For i in range(0, MAX_CODE_SIZE):
        Set the ith element of the bits array to 0.

FUNCTION code_size:
    Return the index of top.

FUNCTION code_empty:
    IF top is equal to 0:
        Return True.
    Return False.

FUNCTION code_full:
    IF top is equal to MAX_CODE_SIZE:
        Return True.
    Return False.

FUNCTION code_set_bit:
    IF i is negative or i is greater than MAX_CODE_SIZE – 1:
        Return False.
    Set the ith element in the bits array to 1.
    Return True.

FUNCTION code_clr_bit:
    IF i is negative or i is greater than MAX_CODE_SIZE – 1:
        Return False.
    Set the ith element in the bits array to 0.
    Return True.

FUNCTION code_get_bit:
    IF i is negative or i is greater than MAX_CODE_SIZE – 1:
        Return False.
    IF the ith index of the bits array is equal to 1:
        Return True.
    Return False.

FUNCTION code_push_bit:
    IF the bit array is full:
        Return False.
    Set the top index of the bit array to the inputted bit.
    Increment top by 1.
    Return True.
```

```
FUNCTION code_pop_bit:
    IF the bit array is empty:
        Return False.
    Set the top element of the bit array to 0.
    Decrement top by 1.

FUNCTION code_print:
    FOR i in range (0, MAX_CODE_SIZE):
        Print the ith element of the array of bits.
```

io.c

```
FUNCTION read_bytes:
    WHILE the return value for read() is greater than 0:
        read() a block of bytes from the input file.
        Store them in buffer.

    Return the number of bytes read from the inputfile

FUNCTION write_bytes:
    WHILE the return value for write() is greater than 0:
        write() a block of bytes to the output file.

    Return the number of bytes read from the output file.

FUNCTION read_bit:
    Store BLOCK number of bytes in the buffer.
    IF read()  < 1:
        Return False.
    Return True.

FUNCTION write_code:
    FOR i in range(0, MAX_CODE_SIZE):
        Add ith index of the Code into the buffer.
    Write() the contents of the buffer to the output file.

FUNCTION flush_code:
    Call write() using the leftover buffered bits.
    Zero out extra bits in the last byte.
```

stack.c

```
FUNCTION stack_create:
    Allocate memory for a stack struct.
    IF the allocation for the stack struct succeeded:
```

```
                Set the top of the stack to index 0.
                Set the capacity of the stack to the inputted capacity.
                Allocate memory for the array of nodes with a size of the inputted capacity.
                IF the memory allocation for the array of nodes succeeded:
                        Return pointer to the Stack.
        Free the memory allocated for the stack struct.
Deference the stack pointer.
Return null pointer.


FUNCTION stack_delete:
IF the pointer to the Stack is not null:
                Deallocate memory for the array of nodes.
                Deallocate the memory for the Stack struct.
                Deference the pointer to the Stack.

FUNCTION stack_empty:
        IF the stack top equals 0:
                Return True.
        Return False.

FUNCTION stack_full:
        IF the stack top is equal to the stack capacity:
                Return True.
        Return False.

FUNCTION stack_size:
        Return index of the stack top.

FUNCTION stack_push:
        IF the stack is full:
                Return False.
        Set the top index of the Stack the inputted node.
        Increment top by 1.
        Return True.

FUNCTION stack_pop:
        IF the stack is empty:
                Return False.
        Set the inputted pointer to the value of the top index of the stack.
        Decrement top by 1.
        Return True.

FUNCTION stack_print:
        FOR each node in the stack:
                Print symbol and frequency.
```

huffman.c

```
FUNCTION build_tree:
    Create a priority queue with size ALPHABET
    For i in range(0, ALPHABET):
        IF the ith index of the histogram is greater than 0:
            Create a node with symbol as i and frequency to the value of ith index.
            Enqueue the node into the priority queue.

    WHILE size of the priority queue is greater than 1:
        Dequeue a node to be the left child.
        Deque a node to be the right child.
    Join  the left and right child nodes using node_join().
    Enqueue the parent node.
Return the one node that is left in the priority queue which is the root of the Huffman tree.

FUNCTION build_codes:
    Create a new Code using code_init().
    IF node is not null:
        IF left node and right node are null:
            Set Code index of the current node to code.
    ELSE:
        Push a 0 into the code.
        Call build_code with the left node.
        Pop a bit from the code.

        Push 1 into the code.
        Call build_code with the right node.
        Pop a bit from the code.

FUNCTION dump_tree:
    IF the root is not null:
        Call dump_tree using the left child of the root.
        Call dump_tree using the right child of the root.
    IF left and right child nodes are null:
        Write ‘L’ .
        Write the symbol of the node.

    ELSE:
        Write ‘I’.

FUNCTION rebuild_tree:
    Create a stack of size nbytes using stack_create.
    FOR i in range(0, nbytes):
        IF the ith element of the array is equal to ‘L’:
            Call node_create() using the symbol from the i+th element.
            Push the created node onto the stack.

        IF the elemnt of the array is equal to ‘I’:
            Pop the stack to get the left child.
            Pop the stack to get the right child.
            Join the left and right children using node_join().

    Pop the last node in the stack which is the root node.
    Return the root node.

FUNCTION delete_tree:
    IF node is null:
        Return.
    Call delete_tree using left child node.
    Call delete_tree using right child node.
    Delete the node using node_delete.
```

Error Handling:
Check that the magic number is valid.