CSE 13S Fall 2021
James Gu
jjgu@ucsc.edu
28 October 2021

Assignment 5: Huffman Coding

Description of Program:

In computer science, data compression is a very commonly used technique that's used in numerous cases like in files, zips, and forms of communication between two different machines. So for the goal of this assignment, we have to implement both an encoder and decoder. The encoder involves creating a histogram of the given file, constructing a Huffman tree , and based on this tree encode it to an output file. The decoder reads the compressed file and decompresses it so it is back to the original input file.

Files:

encode.c:
This file will contain your implementation of the Huffman encoder.

decode.c:
This file will contain your implementation of the Huffman decoder.

defines.h:
This file will contain the macro definitions used throughout the assignment. You may not modify this file.

header.h:
This will contain the struct definition for a file header. You may not modify this file.

node.h:
This file will contain the node ADT interface. This file will be provided. You may not modify this file.

node.c:
This file will contain your implementation of the node ADT.

pq.h:
This file will contain the priority queue ADT interface. This file will be provided. You may not modify this file.

pq.c:
This file will contain your implementation of the priority queue ADT. You must define your priority queue struct in this file.

code.h:
This file will contain the code ADT interface. This file will be provided. You may not modify this file.

code.c:
This file will contain your implementation of the code ADT.

io.h:
This file will contain the I/O module interface. This file will be provided. You may not modify this file.

io.c:
This file will contain your implementation of the I/O module.

stack.h:
This file will contain the stack ADT interface. This file will be provided. You may not modify this file.

stack.c:
This file will contain your implementation of the stack ADT. You must define your stack struct in this file.

huffman.h:
This file will contain the Huffman coding module interface. This file will be provided. You may not modify this file.

huffman.c:
This file will contain your implementation of the Huffman coding module interface.

Makefile: This is a file that will allow the grader to type make to compile your programs.
• CC = clang must be specified.
• CFLAGS = -Wall -Wextra -Werror -Wpedantic must be included.
• make should build the encoder and the decoder, as should make all.
• make encode should build just the encoder.
• make decode should build just the decoder.
• make clean must remove all files that are compiler generated.
• make format should format all your source code, including the header files.

README.md:
This must be in Markdown. This must describe how to build and run your program.

DESIGN.pdf:
This must be a PDF. The design document should answer the pre-lab questions, describe the purpose of your program, and communicate its overall design with enough detail such that a

sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. C code is not considered pseudocode.

Pseudocode:

```
encode.c:
Step 1 Compute histogram:
Read from the file that was given from the user.
Construct histogram:
    Process each line from the file and tally up each occurrence of each unique symbol.
Step 2 Construct Huffman tree:
min_child(A, first, last):
    Set left to 2*first
    Set right to left + 1
    If right is greater than or equal to last and the right -1th element of A is less than the
    left-1th element of A:
        Return right
    Return left

fix_heap(A, first, last):
    Set found to false
    Set mother to first
    Set least to min_child(A, mother, last)
    While mother is greater than or equal to last/2 and found is false:
        If the mother-1th element of A is greater than the least-1th element of A:
            Set temp to the mother-1th element of A
            Set the mother-1th element of A to the least-1 th element of A
            Set the least-1th element of A to temp
            Set mother to least
            Set least to min_child(A, mother, last)
        Else:
            Set found to true

Form Huffman tree:
    Set father to last/2
    For when father is still less than first - 1:
        fix_heap(A, father, last)
        Subtract 1 from father

While the tree size > 1:
    Set left to dequeue()
    Set right to dequeue()
    Set parent to join(left, right)
    enqueue(parent)
```

```
Set root to dequeue()

Step 3 Walk Huffman tree to construct the corresponding code each symbol:
While the tree size > 1:
    Go to the left node and add a 0 to the code
    If the current node has no children:
        Mark current leaf as visited
        Back track to beginning
    If the current node’s left leaf is visited, go right:
        Add 1 to the code.
        Back track to beginning

Step 4 Dump the Tree:
postorder(n):
    If node is not NULL:
        postorder(n’s left)
        postorder(n’s right)
        If node is a leaf add L + symbol to the code
        If node is an interior node add I to the code

Step 5:
Add the encoded code from step 3 to the end of the string from the dumped tree from step 4

decode.c:
Step 1 Reconstruct the Huffman tree from the input dumped tree:
Loop through the tree dump until done:
    Set Right to pop()
    Set Left to pop()
    Parent = join(left, right)
    push(Parent)

Step 2 Walk the bits, traverse the tree:
Loop through the encoded bit code string:
    If the char is ‘0’:
        Go through the left path of the tree
    If the char is ‘1’:
        Go through the right path of the tree
    If the current node is leaf:
        Add the leaf’s symbol to the output string
        Back track to start of the tree

node.c:
Struct Node contains:
    *left of type Node
    *right of type Node
    symbol of type uint8_t
    frequency of type uint64_t

Node *node_create(uint8_t symbol, uint64_t frequency):
    Set up Node *n.
    Set n’s symbol to symbol
    Set n’s frequency to frequency
    Return n

Void node_delete(Node **n):
    Free *n
    Set *n to NULL

Node *node_join(Node *left, Node *right):
    Set up Node *parent
    Set parent’s left to left
    Set parent’s right to right
    Set parent’s frequency to left’s frequency + right’s frequency
    Set parent’s symbol to ‘$’
    Return parent

pq.c:
Struct PriorityQueue contains:
    Top of type uint32_t
    *queue of type Node
    Capacity of type uint32_t

PriorityQueue *pq_create(uint32_t capacity):
    Set up PriorityQueue q
    Set q’s top to 0
    Set q’s queue to an array of all 0’s
    Set q’s capacity to capacity
    Return q

Void pq_delete(PriorityQueue **q):
    Free *q
    Set *q to NULL

Bool pq_empty(PriorityQueue *q):
    If q’s top is 0:
        Return true
    Return false
```

```
Bool pq_full(PriorityQueue *q):
        If q’s queue[capacity] is not 0:
                Return true:
        Return false

Uint32_t pq_size(PriorityQueue *q):
        Return q’s top-1;

Bool enqueue(PriorityQueue *q, Node *n):
        if(pq_full):
                Return false
        Add n to q’s queue
        Add 1 to q’s top
        fix_heap
        Return true

Bool dequeue(PriorityQueue *q, Node **n):
        if(pq_empty):
                Return false
        Set *n to q’s queue[q’s top]
        Set q queue[q’s top] to 0
        Subtract 1 from q’s top
        Return true

defines.h:
Define BLOCK as 4096
Define AlLPHABET as 256
Define MAGIC 0xBEEFDOOD
Define MAX_CODE_SIZE (ALPHABET / 8)
Define MAX_TREE_SIZE = (3*ALPHABET -1)

code.c:
Struct Code contains:
        Top of type uint32_t
        bits[MAX_CODE_SIZE] of type uint8_t

Cede_init(void):
        Set up c of type Code
        Set c’s top to 0
        Set c’s bits[] to all 0’s
        Return c

Bool code_empty(Code *c):
        If c’s top is 0:
```

```
        Return true
    Return false

Bool code_full(Code *c):
    If c’s top-1 is equal to MAX_CODE_SIZE:
        Return true
    Return false

Bool code_set_bit(Code *c, uint32_t i):
    If i is greater than MAX_CODE_SIZE:
        Return false
    Set c’s bits[i] to 1
    Return true

Bool code_clr_bit(Code *c, uint32_t i):
    If i is greater than MAX_CODE_SIZE:
        Return false
    Set c’s bits[i] to 0
    Return true

Bool code_get_bit(Code *c, uint32_t i):
    If i is greater than MAX_CODE_SIZE or c’s bits[i] is 0:
        Return false
    Return true

Bool code_push_bit(Code *c, uint8_t bit):
    If(code_full):
        Return false
    Add bit to c’s bits[]
    Add 1 to top
    Return true

Bool code_pop_bit(Code *c, uint8_t bit):
    If(code_empty):
        Return false
    Set bit to c’s bits[top-1]
    Set c’s bits[top-1] to 0
    Subtract 1 from top
    Return true
```

i/o and huffman coding module are explained in the steps in encode.c and decode.c

stack.c:
struct Stack contains:

```
        top of type uint32_t
        capacity of type uint32_t
        *items of type Node

Stack *stack_create(uint32_t capacity):
        Set up a Stack called s
        If s doesn’t need to be freed:
                Set s’s top to 0
                Set s’s capacity to the value of capacity
                Set s’s items to an array of all 0’s
                If s needs to be freed:
                        Free s
                        Set s to NULL
Return s

Void stack_delete(Stack **s):
        If *s and s’s items need to be freed:
                Free s’s items
                Free s
        Return

Bool stack_empty(Stack *s):
        If s’s items array is empty:
                Return true
        Return false

Bool stack_full(Stack *s):
        If the s’s items[capacity] is not 0;
                Return true
        Return false

uint32_t stack_size(Stack *s):
        Set size to 0
        Set i to 0
        For the elements of s’s items array:
                If items[i] is not 0:
                        Add 1 to size
                Else:
                        Break
                Add one to i
        Return size

Bool stack_push(Stack *s, Node *n):
        If stack_full(s) is true:
```

```
        Return false
    Add n to s’s items[s’s top]
    Add one to set’s top
    Return true

Bool stack_pop(Stack *s, Node **n):
    If stack_empty(s) is true:
        Return false
    Set *n to s’s items[s’s top]
    Set s’s items[s’s top] to 0
    Subtract 1 from s’s top
    Return true
```