Assignment 8 – Huffman coding

Vybavnag Kandasamy

CSE 13S – Fall 2023

Purpose

The Huffman Coding program is designed for data compression. It achieves this by analyzing the frequency of different bytes in a data file and then creating a compressed version of the file using fewer bits for more frequent bytes and more bits for less frequent ones. The program comprises two parts: a compressor, huff, which generates the Huffman code for an input file, and a decompressor, dehuff, which restores the original file from its Huffman encoded version.

How to Use the Program

Ensure all files are present. To use the Huffman Coding program, begin by compiling the program using make. Once compiled, to compress a file, use the command ./huff -i [inputfile] -o [outputfile], where -i specifies the input file and -o the output file. For decompression, the command ./dehuff -i [inputfile] -o [outputfile] is used, where the input file is the Huffman encoded file and the output file is the decompressed original file. Make clean to remove excutables and make format to format the code. The program also include a -h flag for help.

```
make (make all)
make format
./huff -i [inputfile] -o [outputfile]
./dehuff -i [inputfile] -o [outputfile]
make clean
```

Program Design

In the Huffman Coding project, the required files are organized to encompass various aspects of the program. The project includes several provided header files, such as bitreader.h, bitwriter.h, node.h, and pq.h, each serving as a blueprint for crucial components. The bitreader.h and bitwriter.h files define the structure and functions for reading and writing bits. The node.h file outlines the structure of the nodes used in building the Huffman tree. Similarly, pq.h is dedicated to the Priority Queue.

There are also unit-test files: brtest.c, bwtest.c, nodetest.c, and pqtest.c. Each of these is tailored to test the corresponding components.

This project also includes two key executable files, huff.c and dehuff.c. The huff.c file is responsible for implementing the Huffman compression algorithm, turning standard files into compressed versions. On the other hand, dehuff.c serves the opposite purpose, decompressing files that were previously compressed using the Huffman method.

A Makefile is also an essential part of the submission, containing instructions for compiling and managing the various components of the project. The project also contains this report that details everything out.

Data Structures

First, there are BitWriter and BitReader. Think of them as helpers that deal with tiny bits of data in files. They're super important for changing regular data into a special kind of code, known as Huffman code. This code is like a secret language that uses shorter symbols for things that show up a lot and longer symbols for rarer things in the file.

Then, there's a part called Node. It helps build a tree-like structure. Each Node carries a piece of information from the file (like a letter or a number) and counts how often it appears. This tree is like a map that shows how to turn the file into this new secret language.

Also, the program has something called a Priority Queue. You can think of it as a smart line where each Node waits. This line is special because it puts the most common pieces at the front. This helps the program decide which pieces should get the shorter symbols in the Huffman code.

Algorithms

Pusdeocode listed below.

Bitreader

```
  /* bitreader.h */
  typedef struct BitReader BitReader;

  /* bitreader.c */
  #include "bitreader.h"
  struct BitReader {
      FILE *underlying_stream;
      uint8_t byte;
      uint8_t bit_position;
    };

BitReader *bit_read_open(const char *filename){
    allocate a new BitReader
    open the filename for reading as a binary file, storing the result in FILE *f
    store f in the BitReader field underlying_stream
    clear the byte field of the BitReader to 0
    set the bit_position field of the BitReader to 8
    if any step above causes an error:
    return NULL
    else:
        return a pointer to the new BitReader
}

void bit_read_close(BitReader **pbuf){
  if *pbuf != NULL:
      close the underlying_stream
      free *pbuf
      *pbuf = NULL
      if any step above causes an error:
          report fatal error
}

uint8_t bit_read_bit(BitReader *buf){
    if bit_position > 7:
        read a byte from the underlying_stream using fgetc()
        bit_position = 0
    get the bit numbered bit_position from byte
    bit_position += 1;
    if any step above causes an error:
        report fatal error
```

```
    else:
        return the bit
}

uint8_t bit_read_uint8(BitReader *buf){
    uint8_t byte = 0x00
    for i in range(0, 8):
        read a bit b from the underlying_stream
        set bit i of byte to the value of b
    return byte
}

uint16_t bit_read_uint16(BitReader *buf){
   uint16_t word = 0x0000
   for i in range(0, 16):
      read a bit b from the underlying_stream
      set bit i of word to the value of b
  return word;
}

uint32_t bit_read_uint32(BitReader *buf, uint32_t x){
     uint32_t word = 0x00000000
     for i in range(0, 32):
      read a bit b from the underlying_stream
      set bit i of word to the value of b
    return word;
}
```

Bitwrite

```
/* bitwriter.h */
      typedef struct BitWriter BitWriter;

/* bitwriter.c */
#include "bitwriter.h"
struct BitWriter {
    FILE *underlying_stream;
    uint8_t byte;
    uint8_t bit_position; /*
};

BitWriter *bit_write_open(const char *filename){
    allocate a new BitWriter
    open the filename for writing as a binary file, storing the result in FILE *f
    store f in the BitWriter field underlying_stream
    clear the byte and bit_positions fields of the BitWriter to 0
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new BitWriter
}

void bit_write_close(BitWriter **pbuf){
    if *pbuf != NULL:
      if (*pbuf)->bit_position > 0:
          /* (*pbuf)->byte contains at least one bit that has not yet been written */
          write the byte to the underlying_stream using fputc()
      close the underlying_stream
      free the BitWriter
      *pbuf = NULL
```

```
}

void bit_write_bit(BitWriter *buf, uint8_t x){
       if bit_position > 7:
           write the byte to the underlying_stream using fputc()
           clear the byte and bit_position fields of the BitWriter to 0
    set the bit_position bit of the byte to x
    bit_position += 1
}

void bit_write_uint8(BitWriter *buf, uint8_t x){
    for i = 0 to 7:
      write bit i of x using bit_write_bit()
}

void bit_write_uint16(BitWriter *buf, uint16_t x){
     for i = 0 to 15:
      write bit i of x using bit_write_bit()
}

void bit_write_uint32(BitWriter *buf, uint32_t x){
    for i=0to31:
      write bit i of x using bit_write_bit()
}
```

Node

```
/* node.h */
      typedef struct Node Node;
      struct Node {
          uint8_t symbol;
          uint32_t weight;
          uint64_t code;
          uint8_t code_length;
          Node *left;
          Node *right;
      };

Node *node_create(uint8_t symbol, uint32_t weight){
    allocate a new Node
    set the symbol and weight fields of Node to function parameters symbol and weight
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new Node
}

void node_free(Node **node){
    if *pnode != NULL:
      free(*pnode)
      *pnode = NULL
}
void node_print_tree(Node *tree, char ch, int indentation){
       void node_print_node(Node *tree, char ch, int indentation) {
       if (tree == NULL)
           return;
       node_print_node(tree->right, '/', indentation + 3);
       printf("%*cweight = %.0f", indentation + 1, ch, tree->weight);
       if (tree->left == NULL && tree->right == NULL) {
           if (' ' <= tree->symbol && tree->symbol <= '~') {
```

```
                printf(", symbol = '%c'", tree->symbol);
            } else {
                printf(", symbol = 0x%02x", tree->symbol);
            }
        }
            printf("\n");
            node_print_node(tree->left, '\\', indentation + 3);
        }

        void node_print_tree(Node *tree) {
            node_print_node(tree, '<', 2);
        }
    }
```

Priority Queue

```
/* pq.h */
     typedef struct PriorityQueue PriorityQueue;

 /* pq.c */
     typedef struct ListElement ListElement;
     struct ListElement {
         Node *tree;
         ListElement *next;
     };
     struct PriorityQueue {
         ListElement *list;
    };

    PriorityQueue *pq_create(void){
        Allocate memory for a new PriorityQueue object.
        If memory allocation is successful:
        Set the list field of the PriorityQueue to NULL.
        Return the pointer to the newly created PriorityQueue.
        If memory allocation fails, return NULL.
    }

    void pq_free(PriorityQueue **q){
        Check if the pointer to the PriorityQueue and the PriorityQueue itself are not NULL.
        Free the PriorityQueue object.
        Set the pointer to the PriorityQueue to NULL.
    }

    bool pq_is_empty(PriorityQueue *q){
        Return true if the PriorityQueue is NULL or its list field is NULL.
        Otherwise, return false.
    }

    bool pq_size_is_1(PriorityQueue *q){
       Return true if the PriorityQueue is not NULL and its list has exactly one element (i.e., list
           ->next is NULL).
       Otherwise, return false.
    }

    bool pq_less_than(ListElement *e1, ListElement *e2){
        Compare two ListElement objects based on the weight in their Node.
        If the first Node's weight is less than the second Node's weight, return true.
        If weights are equal, compare their symbol values and return true if the first is less.
        Otherwise, return false.
    }
```

```
void enqueue(PriorityQueue *q, Node *tree){
    enqueue(node, tree):
    Allocate ListElement new_element
    set the tree field to the value of the tree function parameter
    if the queue is empty:
       point the queue to new_element
    elif pq_less_than(new_element, q->list):
       New element E1 goes before all existing elements of the list

     Q -----> E2 --> E2 ...
         ^
         E1

     Q --> E1 --> E2
                  ^
                  E3
     Or the new element E3 goes before an existing element E4:
     Q --> E1 --> E2 -----> E4 ...
                     ^
                     E3
    In both cases, we are looking for existing element E2,
       insert new element E1 as the first element of the list
    else:
    Either the new element E3 goes at the end of the list: */
    and then we put the new element E3 after it. */
    find existing element E2 (either E2->next is NULL or E3 < E2->next)
    insert new element after E2

}

Node *dequeue(PriorityQueue *q){
    If the queue is empty, print an error message and exit.
    Otherwise, remove the first element from the list (the one with the lowest weight).
    Save the Node from this element.
    Free the removed list element.
    Return the saved Node.
}

void pq_print(PriorityQueue *q){
       assert(q != NULL);
         ListElement *e = q->list;
         int position = 1;
         while (e != NULL) {
            if (position++ == 1) {
    printf("============================================\n"); }else{
                printf("--------------------------------------------\n");
            }
            node_print_tree(e->tree, '<', 2);
    e = e->next; }
}
```

Huffman

```
typedef struct Code {
uint64_t code;
uint8_t code_length;
} Code;
```

```
uint32_t fill_histogram(FILE *fin, uint32_t *histogram){
       histogram = [0] * 256
       filesize = 0

       with open(file_path, 'rb') as fin:
           while True:
               byte = fin.read(1)
               if not byte:
                   break

               histogram[byte[0]] += 1
               filesize += 1

       histogram[0x00] += 1
       histogram[0xff] += 1

       return filesize, histogram
   }

Node *create_tree(uint32_t *histogram, uint16_t *num_leaves){
   pq = PriorityQueue()
   num_leaves = 0


   for symbol, weight in enumerate(histogram):
       if weight > 0:
           pq.enqueue(Node(symbol, weight))
           num_leaves += 1


   while not pq.size_is_one():
       left = pq.dequeue()
       right = pq.dequeue()

       merged_node = Node(None, left.weight + right.weight)
       merged_node.left = left
       merged_node.right = right

       pq.enqueue(merged_node)

   return pq.dequeue(), num_leaves
}

fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length){
   if node is internal:
   /* Recursive calls left and right. */
   /* append a 0 to code and recurse *
   /* (don't neeed to append a 0; it's already there)
   fill_code_table(code_table, node->left, code, code_length + 1);
   /* append a 1 to code and recurse */
   code |= (uint64_t) 1 << code_length;
   fill_code_table(code_table, node->right, code, code_length + 1);
   else:
   /* Leaf node: store the Huffman Code. */
   code_table[node->symbol].code = code;
   code_table[node->symbol].code_length = code_length;
}
```

```
void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table){
    write uint8_t 'H' to outbuf
    write uint8_t 'C' to outbuf
    write uint32_t filesize to outbuf
    write uint16_t num_leaves to outbuf
    huff_write_tree(outbuf, code_tree)
    while true:
    b = fgetc(fin)
    if b == EOF:
    break
    code = code_table[b].code
    code_length = code_table[b].code_length
    for i in range(0, code_length):
    write bit (code & 1) to outbuf
    code >>= 1
}
```

dehuff

```
void dehuff_decompress_file(FILE *fout, BitReader *inbuf){
    read uint8_t type1 from inbuf
    read uint8_t type2 from inbuf
    read uint32_t filesize from inbuf
    read uint16_t num_leaves from inbuf
    assert(type1 == 'H')
    assert(type2 == 'C')
    num_nodes = 2 * num_leaves - 1
    Node *node
    for i in range(0, num_nodes):
    read one bit from inbuf
    if bit == 1:
    read uint8_t symbol from inbuf
    node = node_create(symbol, 0)
    else:
    node = node_create(0, 0)
    node->right = stack_pop()
    node->left = stack_pop()
    stack_push(node)
    Node *code_tree = stack_pop()
    for i in range(0, filesize):
    node = code_tree
    while true:
    read one bit from inbuf
    if bit == 0:
    node = node->left
    else:
    node = node->right
    if node is a leaf:
    break
    write uint8 node->symbol to fout
    }
```

Function Descriptions

BitReader: BitReader *bit_read_open(const char *filename)

- Inputs: Filename
- Outputs: Returns a pointer to a BitReader

- Purpose: To open the file and return a pointer to bit reader .
- Description: Opens a file specified by the filename and initializes a BitReader structure. It allocates memory for the BitReader and prepares it for reading bits from the file, handling any necessary file access and buffering setup. The function returns a pointer to the BitReader if successful, or NULL if the file can't be opened or if there's a memory allocation failure.

Void bit_read_close(BitReader **pbuf)

- Inputs: Pointer to bitreader.
- Outputs: NAN
- Purpose: To the free the pointer and set *pbuf pointer to NULL.
- Description: Closes the bit reader, ensuring that any associated resources, such as file handles and allocated memory, are released. It sets the pointer to the BitReader (pointed to by pbuf) to NULL to prevent any accidental use after deallocation.

uint8_t bit_read_bit(BitReader *buf)

- Inputs: Pointer to bitreader.
- Outputs: Returns single bit vaules of what the pointer points to.
- Purpose: Reads a single bit using values in the BitReader pointed to by buf.
- Description: Reads a single bit from the BitReader pointed to by buf. It handles the extraction of the bit from the current position in the file or buffer and advances the read position accordingly.

uint8_t bit_read_uint8(BitReader *buf)

- Inputs: Pointer to bitreader.
- Outputs: Returns 8 bits by calling read bit 8 times.
- Purpose: Reads 8 bits from buf by calling bit_read_bit() 8 times. Collect these bits into a uint8_t starting with the LSB .
- Description: : Reads 8 bits sequentially from the BitReader and assembles them into a single byte (uint8_t). It starts with the least significant bit, combining the bits in order to form the byte.

uint16_t bit_read_uint16(BitReader *buf)

- Inputs: Pointer to bitreader.
- Outputs: Returns 16 bits by calling read bit 16 times.
- Purpose: Similar to bit_read_uint8, this function reads 16 bits from the BitReader and assembles them into a 16-bit unsigned integer. It starts from the least significant bit and works up to the most significant bit.

uint32_t bit_read_uint32(BitReader *buf, uint32_t x)

- Inputs: Pointer to bitreader, unit32 varaible.
- Outputs: Returns 32 bits by calling read bit 32 times.
- Purpose: Reads 32 bits of the uint32_t x by calling bit_read_bit() 32 times. Start with the LSB.
- Description: Reads 32 bits from the BitReader, combining them into a 32-bit unsigned integer. The reading process begins from the least significant bit of the uint32_t x and proceeds to the most significant bit, assembling the full 32-bit value.

BitWrite: BitWriter *bit_write_open(const char *filename)

- Inputs: Filename.
- Outputs: Returns a pointer to a BitWriter.
- Purpose: Open binary filename for write using fopen() and return a pointer to a BitWriter.
- Description: This function creates a new BitWriter object. It opens a file specified by the filename in binary write mode. If successful, it returns a pointer to the initialized BitWriter; otherwise, it returns NULL.

void bit_write_close(BitWriter **pbuf)

- Inputs: Pointer to Bitwriter.
- Outputs: Frees the pointer and set *pbuf pointer to NULL.
- Purpose: Close underlying_stream, free the BitWriter object, and set the *pbuf pointer to NULL.
- Description: This function is responsible for properly closing and cleaning up a BitWriter object. It closes the file stream associated with the BitWriter, frees the memory allocated to the BitWriter, and sets the pointer referenced by pbuf to NULL.

void bit_write_bit(BitWriter *buf, uint8_t x)

- Inputs: Pointer to Bitwriter, uint8 variable.
- Outputs: Single bit x value.
- Purpose: It writes a single bit, x, using values in the BitWriter pointed to by buf. This function collects 8 bits into the buffer byte before writing it using fputc().
- Description: This function writes a single bit to the BitWriter. It buffers the bit until 8 bits are collected, then writes these as a byte to the file. The x parameter should be either 0 or 1.

void bit_write_uint8(BitWriter *buf, uint8_t x)

- Inputs: Pointer to Bitwriter, uint8 variable.
- Outputs: 8 bits x value.
- Purpose: Write the 8 bits of function parameter x by calling bit_write_bit() 8 times. Start with the LSB.
- Description: This function writes an 8-bit unsigned integer to the BitWriter. It breaks down the integer x into individual bits and writes them sequentially, starting from the least significant bit (LSB).

void bit_write_uint16(BitWriter *buf, uint16_t x)

- Inputs: Pointer to Bitwriter, uint16 variable.
- Outputs: 16 bits x value.
- Purpose: Write the 16 bits of function parameter x by calling bit_write_bit() 16 times. Start with the LSB.
- Description: This function handles the writing of a 16-bit unsigned integer. It iteratively writes each bit of x to the BitWriter, beginning with the LSB, by calling bit_write_bit for each bit.

void bit_write_uint32(BitWriter *buf, uint32_t x)

- Inputs: Pointer to Bitwriter, uint32 variable.

- Outputs: 32 bits x value.
- Purpose: Write the 32 bits of function parameter x by calling bit_write_bit() 32 times. Start with the LSB.
- Description: This function is designed for writing a 32-bit unsigned integer. It sequentially writes each bit of the integer x, starting from the LSB, to the BitWriter using the bit_write_bit function.

Node: Node *node_create(uint8_t symbol, uint32_t weight)

- Inputs: Symbol and Weight of Node.
- Outputs: Pointer to new node.
- Purpose: Create a Node and set its symbol and weight fields. Return a pointer to the new node.
- Description: This function dynamically allocates memory for a new Node and initializes its 'symbol' and 'weight' fields with the provided arguments.

void node_free(Node **node)

- Inputs: Node Pointer.
- Outputs: Frees the pointer and sets *node to null.
- Purpose: To free the pointer and set it to NULL.
- Description: This function safely frees a Node pointed to by the provided pointer. It checks if the pointer is not NULL, frees the allocated memory, and sets the pointer to NULL to prevent dangling pointer issues.

void node_print_tree(Node *tree, char ch, int indentation)

- Inputs: Node *tree (root node of the binary tree), char ch (branch direction character), int indentation (indentation level).
- Outputs: Visualisation of the Binary tree.
- Purpose:This function, through recursive traversal, prints each node of a binary tree with indentation indicating depth and characters showing branch directions. Nodes display their weight, and leaf nodes also show their symbol (as a character or in hexadecimal).

Priotity Queue:
PriorityQueue *pq_create(void)

- Inputs: NAN.
- Outputs: Returns A pointer to a newly created PriorityQueue object.
- Purpose: To create and initialize a new priority queue.
- Description: This function allocates memory for a new PriorityQueue object. If memory allocation is successful, it initializes the list field to NULL and returns the pointer to the PriorityQueue. If memory allocation fails, it returns NULL.

void pq_free(PriorityQueue **q)

- Inputs: A pointer to a PriorityQueue object.
- Outputs: NAN.
- Purpose: To free the memory allocated for a priority queue.

- Description: This function checks if the pointer to the PriorityQueue and the PriorityQueue itself are not NULL. If they are not, it frees the PriorityQueue object and sets the pointer to NULL to prevent dangling references.

bool pq_is_empty(PriorityQueue *q)

- Inputs: A pointer to a PriorityQueue object.
- Outputs: A boolean value.
- Purpose: To check if the priority queue is empty.
- Description: Returns true if the PriorityQueue is not NULL and its list has exactly one element (i.e., list-¿next is NULL). Otherwise, it returns false.

bool pq_size_is_1(PriorityQueue *q)

- Inputs: A pointer to a PriorityQueue object.
- Outputs: A boolean value.
- Purpose: To check if the priority queue contains exactly one element.
- Description: Returns true if the PriorityQueue is not NULL and its list has exactly one element (i.e., list-¿next is NULL). Otherwise, it returns false.

bool pq_less_than(ListElement *e1, ListElement *e2)

- Inputs: Two pointers to ListElement objects.
- Outputs: A boolean value.
- Purpose: To compare two ListElement objects based on the weights and symbols in their Node.
- Description: This function compares two ListElement objects based on the weight in their Node. If the first Node's weight is less than the second Node's weight, or if weights are equal and the first symbol is less, it returns true. Otherwise, it returns false.

void enqueue(PriorityQueue *q, Node *tree)

- Inputs: A pointer to a PriorityQueue object and a pointer to a Node object.
- Outputs: NAN.
- Purpose: To add a new element to the priority queue.
- Description: This function allocates a new ListElement and sets its tree field. If the queue is empty, the new element becomes the first element. If the new element has a lower weight than the first element in the queue, it is inserted at the beginning. Otherwise, it is inserted at the appropriate position based on its weight.

Node *dequeue(PriorityQueue *q)

- Inputs: A pointer to a PriorityQueue object.
- Outputs: NAN.
- Purpose: A pointer to a Node object.
- Description: f the queue is not empty, this function removes the first element from the list (the one with the lowest weight), saves the Node from this element, frees the removed list element, and returns the saved Node. If the queue is empty, it prints an error message and exits.

void pq_print(PriorityQueue *q)

- Inputs: A pointer to a PriorityQueue object.
- Outputs: NAN.
- Purpose:To print the elements of the priority queue.
- Description: This function asserts that the queue is not NULL, then iterates through each ListElement in the queue's list. It prints the tree structure of each Node in the list elements, providing a visual representation of the priority queue.

Huffman: uint32_t fill_histogram(FILE *fin, uint32_t *histogram)

- Inputs: A file pointer to the input file, A pointer to the histogram.
- Outputs: A tuple containing filesize and an array histogram..
- Purpose: To read a file byte-by-byte and build a histogram of byte frequencies, which is essential for Huffman coding.
- Description: This function iterates over each byte in the file, incrementing the count of each byte value in the histogram array. It also keeps track of the file size. The histogram is adjusted for values 0x00 and 0xFF.

Node *create_tree(uint32_t *histogram, uint16_t *num_leaves)

- Inputs: A pointer to the histogram, A pointer to the number of leaves.
- Outputs: A pointer to the root node of the Huffman tree and the number of leaves in the tree.
- Purpose: To create a Huffman tree based on the frequencies of different byte values in the file.
- Description: This function initializes a priority queue and enqueues nodes for each symbol with a non-zero weight. It then iteratively merges the two nodes with the lowest weight until only one node remains, which becomes the root of the Huffman tree.

fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)

- Inputs: An array of Code structures, a pointer to the current node in the Huffman tree, the current Huffman code, the length of the current Huffman code.
- Outputs: NAN.
- Purpose: To traverse the Huffman tree and assign Huffman codes to each symbol.
- Description: This recursive function traverses the Huffman tree. At each leaf node, it stores the Huffman code and its length in the code table.

void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)

- Inputs: A buffer to write the compressed data, a file pointer to the input file, the size of the input file, the number of leaves in the Huffman tree, the Huffman tree and the table of Huffman codes.
- Outputs: Compressed file data written to outbuf.
- Purpose: To compress a file using Huffman coding.
- Description: This function writes the file header and Huffman tree to the output buffer. It then reads the input file byte-by-byte, retrieves the corresponding Huffman code from the code table, and writes this code to the output buffer.

dehuff:

- Inputs: Pointer to the output file for decompressed data,Pointer to read compressed data bit by bit.

- Outputs: The function writes the decompressed data into the file pointed to by fout.
- Purpose: To decompress a file that has been compressed using Huffman coding.
- Description: The function reads header information (file type, size, number of leaves) from inbuf, checks file format, and constructs a Huffman tree. It then decompresses the data by reading bits, traversing the Huffman tree to decode symbols, and writing them to fout. The process continues until the entire file is decompressed. This function is essential for decompressing data compressed with Huffman coding.

Testing

The testing for this assignment is mainly done through the shell given to me as well as the test files in the assignment repository. These wont get all the test cases so I will be modifying them to get everything. I will also be running Valgrid to check for memory leaks and making sure everything works properly.

Results

Still need to finish code to impliment. 1.

```
veenstra@arm128:~/s23/13s-cse/resources/asgn1$ ./pig_arm
Number of players (2 to 10)? 0
Invalid number of players. Using 2 instead.
Random-number seed? 3
Margaret Hamilton
 rolls 15, has 15
 rolls 5, has 20
 rolls 0, has 20
Katherine Johnson
 rolls 0, has 0
Margaret Hamilton
 rolls 5, has 25
 rolls 0, has 25
Katherine Johnson
 rolls 0, has 0
```

Figure 1: Screenshot of the program running.

References