Mattiwos Belachew
mbelache@ucsc.edu
5/23/2021

CSE 13s Spring 2021
Assignment 6:
Huffman Coding
Design Document

Description:

For this assignment, I will be implementing data compression called Huffman, after it's original creator who helped found the Computer Science Department at UC Santa Cruz. The main key to the data compression is entropy, which is the measure of the amount of information in a set of symbols. Using this information you are able to encode data by assigning the least amount of bits to the most common symbol and the greatest number of bits to the least common symbol thus compressing the original data.

TOP LEVEL:

Encode:

```
encode(){
        //parse program argument, input, output, help,verbose
        File_size = //get input file size
        perm = //get input permission
        //builds histogram
        //creates a histogram and initialize it to 0
        //set the beginning of 0 and 255 of histogram to 1
        //create buffer
        //loop through read_bytes until end of file
                //increase histogram at buffer by 1 //reads the entire file and counts
        frequency
        //counts unique symbols in histogram to use to calculate tree size
        //Builds Tree by called build_tree in huffman.c and give it histogram
        //Creates Code table
        //Call build_codes and give it the table and tree
        //set output permission same as input
        //Create header
        //set header magic to MAGIC value from define.h
        //set header perm to input permission
        //set header file size to input file size
        //set header tree size to (unique symbols /3)-1
        //write the header to the output file
        //call dump tree and give it root and output
        //start reading the input file from the beginning by calling lseek
        //reset bytes_read to 0
        //create a buffer
        //emit code by looping through the input file again one byte at a time
                //write byte the value of histogram at the buffer
        //flush the code
        //if verbose is true then print statistics about encoding
        //close the program
        //Free table
        //delete tree
        //close output file
        //exit successfully
}
```

- The encoded program works by assigning the least number of bits to the most common symbol, and the greatest number of bits to the least common symbol.
- It first counts the frequency that a certain symbol occurs and makes a tree out of it using the build_tree function.
- After it uses the tree it assigns a code to each symbol depending on the location in the tree.
- After it outputs the header containing information about the input and output the tree by dumping it using dump_tree()
- At the end it will use the code table to replace chars in the input with the shortened code (bits) thus compressing the file.

Decode:

```
decode(){
        //parse program argument get input, output, help, verbose, help
        //read header of file (read_bytes)
        //check if the file is valid and my program is able to decode it
        //Set output permissions same as input permission (found on header)
        //rebuild tree from dump
        //creates a buffer of dump tree with the tree size I got from header
        //reads in bytes in order to occupy dump tree
        //call rebuild_tree() to rebuilds original tree made in encode to use
        //creates temp tree in order to decode message later
        //create a buffer to contain the bit read from read_bit
        //create num decoded variable to /keeps track of numbers of chars decoded in
order to end once the file
        //keeps track of numbers of chars decoded in order to end once the file
                //if bit is 0 head to left node
                //if bit is 1 head to the right node
                //if current location is a leaf
                        //write the current leaf symbol output file
                        //increase the number of num decoded to test later
                        //set curr to root
                //if at the end of file end loop (num decoded = header.file_size)
                        //break
        //if verbose is enable prints statistics about the file decoding
        //close program
        //closes the open output file
        //closes the open input file
        //exit successfully
}
```

- The decode reads the header which contains information about the tree size, permission of input, and magic number to determine if it is a valid file which the program can't decode.
- Afterworth, it will read_bytes of the tree size buffer and save it to a variable which will be used to rebuild the tree by iterating over it and reconstructing it.
- After it will use the tree to decode the compressed message after the tree dumps into a valid message by reading it and outputting it to the outfile as the bits are being read.

Huffman:

```
Node *build_tree(uint64_t hist[static ALPHABET]){
        //creates a priority queue
        //loops through histogram
                //if frequency is greater than 1
                        //enqueue the index and symbol at the location
        //loop through priority queue until only one node left
                //get left node
                //get right node
                //join them together and enqueue them
        //remove root from priority queue
        //delete priority queue
        //return root
}
Void build(){
        //create buffer to discards popped code
        //check if root is a leaf
                //save the code to that table symbol
        //else
                //if left root exists
                        //push 0 to the code
                        //explore left node call build()
                        //pop 0 returned from exploring left node
                //if right root exists
                        //push 1 to the code to go right
                        //explore right node call build()
                        //finished exploring left node
        return;
}
void build_codes(Node *root, Code table[static ALPHABET]){
        //initialized new code
        //call build which recursively builds code table
        //return nothing
}
Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes]){
        //creates a stack
        //loops through the tree dump
                //if tree is L
                        //push a node with tree symbol at index to the stack
                        //skip next character
                //if encountered interior node char = I
                        //pop right node
                        //pop left node
                        //join them and push them to the stack
        //pop root node from stack
        //delete stack
        /return node
}
void delete_tree(Node **root){
        //check if root is valid
                //delete left node recursively delete_tree(root->left)
                //delete right node recursively delete_tree(root->right)
                //node delete root
        //return nun;
}
```

- Contains the required function needed to build a tree, build code used to encode the input, and rebuild tree from tree dump. Deletes the tree recursively.

Code:

```
Code code_init(void){
  Code c; //create Code c
  //set top to 0
  //return c
}

Int code_size(Code *c){
   //check if c is valid
   //return c top
}

bool code_empty(Code *c){
   //check if c is valid
   // return true if c top is equal to zero
   //else return false
}

bool code_full(Code *c){
   //check if c is valid
```

```
    // return true if c top is equal to ALPHABET
    //else return false
}

bool code_push_bit(Code *c, uint8_t bit){
    //check if c is valid
    //if c is not full
    //set the top array of c to bit at c top
    //increase c top by 1
    //return true
   //else return false
}

bool code_pop_bit(Code *c, uint8_t *bit){
   //check if c is valid
    //if c is not empty
    //decrease c top by 1
    //set bit address to the the top array of c
    //return true
   //else return false
}

code_print(Code *c){
    //check if c is valid
    for (int i = 0; i < c->top -1;i++){
            printf("%su ", c->bits[i]);
    }
}
```

- Code ADT is used to reconstruct a huffman tree and used to create a code table used to replace the characters in the input into a smaller number of bits.

PriorityQueue:

```
PriorityQueue *pq_create(uint32_t capacity){
   // Create q and allocate enough space for it
   // Check if q was allocated space
   // If so set q head to 0
   //set q tail to 0
   // set q capacity to capacity(parameter)
  // Create q items and allocate enough space for it filled with nodes
  //Check if it didn't work if so free q and set q to NULL
  // return q
}

void pq_delete(PriorityQueue **q){
  //free q items and q and set it to null
  //return nothing
}

bool pq_empty(PriorityQueue *q){
  //check if q exits
  //return true if q size == 0
}

bool pq_full(PriorityQueue *q){i
   //check if q exits
   //return true if q size equal capacity
}

uint32_t pq_size(PriorityQueue *q){
  //check if q exits
  //return q size
}

bool enqueue(PriorityQueue *q, Node *n){
   //check if q exits
   //check if n exits
    uint32_t temp = q->tail;//use modulo function to make it wrap around (q->tail - 1) %
q->capacity)
    while (temp != q->head && (q->items[temp -1]->frequency > n->frequency)){
           //insertion sort
           q->items[temp] = q->items[temp-1]; //shift node
           temp-=1;
    }
    q->items[temp] = n;
    return true;
```

```
}

bool dequeue(PriorityQueue *q, Node **n){
    //check if q exits
    //check if q is not empty
    //decrease q top by 1
    //set n to q items at q top
    return true;
}
void pq_print(PriorityQueue *q){
    //check if q exits
        for (uint32_t i = 0; i < q->size; i++) { //print q values
            printf("%u", ((q->tail + i) % q->capacity));
        }

  //return nothing
}
```

- This is a priority queue of nodes. It is similar to queue where you are able to enqueue and dequeue but the difference is that the enqueues are sorted by frequencies and the dequeues dequeue the lowest frequency every time thus creating a priority queue. The lower the frequency of a node, the higher its priority.

I/O:

```
read_bytes(int infile, uint8_t *buf,nbytes){
   //keeps track of number of bytes read(total)
    //keeps track of bytes read by 1 call of read
    //loop until nbytes have been read or file empty
           ///read infile and add it to bufs;
           //increment total by bytes read
    }
    //increments bytes read by total read this instance
    return total;
}

int write_bytes(int outfile, uint8_t *buf, int nbytes){
   //keeps track of number of bytes read(total)
```

```
    //keeps track of bytes read by 1 call of written
    //loop until nbytes have been written
           ///read infile and add it to bufs;
           //increment total by bytes written
    }
    //increments bytes written by total written this instance
    return total;

}


bool read_bit(int infile, uint8_t *bit) {
    //if the first time bit is called then it fills it
        int bread = read_bytes(infile, buffer, BLOCK);
        //if bytes read is less
            //saves last bit index

    //get bit from buffer at bitindex and set it to *bit
    //increment index
    //check if all bits have been read
        //reset bitindex
    //all bytes/bits have been read in the file
        printf("last bit\n");
        return false;
    return true;
}

void flush_codes(int outfile) {
    //check if there is leftover bytes
           //get remaining bytes in the buffer
           //flush out the leftover bytes
    return;
}
```

- IO contains the interface to interact with input and outfile files in order to output bytes and codes to the output and read bytes and bits as well. This is meant to be a robust I/O module that's used in both encoder and decoder.

Design Process:

1. I started off reimplementing the things I already know how to do and thought about testing each component later one by one to make sure there aren't any problems once they come together.
2. I was confused for a long time why my printtree was different from the test encode but I learned that mine works as well but the difference was that the test encode uses min heap and I use insertion which causes the print tree to act differently. Sucked that I was stuck on it for a few days on it.
3. I wasn't able to figure out how to solve the pipeline problem since O_TMPFILE doesn't work even though it's in the man page of open and it throws errors saying it doesn't exist.

What I learned:

1. I learned how to create a priority queue. I learned from my previous mistakes of implementing queues and fixed them here. Sadly I wasn't able to fix them for the asgn previously since I wasn't aware it was a big problem.
2. I learned how to create a post order traversal and gained confidence about programming it.
3. I learned how to interact with trees and manipulate them since for a long time I was confused on how you go about using trees. But because of the past two assignments it was useful in helping me understand them.
4. I also was able to implement a bit vertex easily this time.

Resources Used:

- Asgn6.pdf
- asgn5Design by William Santosa from Piazza
- Example_design.pdf found in CSE 13s Discord.