Zhongan(Tony) Liu

CSE 13s

03/02/2021

Lempel-Ziv Compression Design

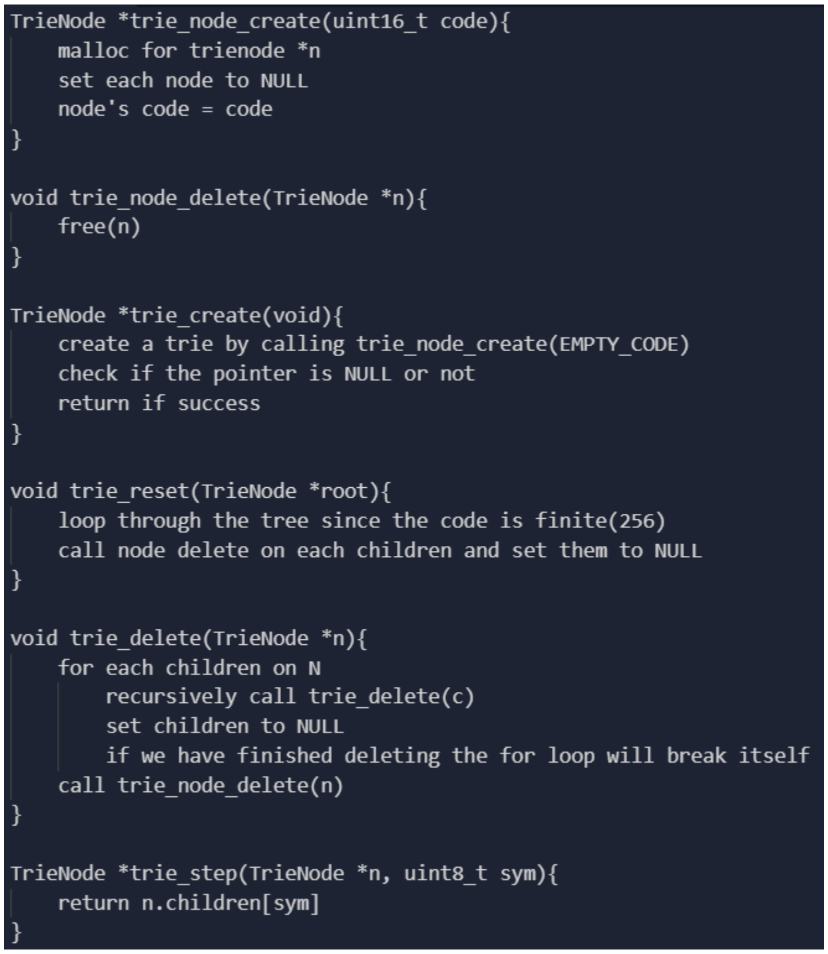
Lempel-Ziv compression is a method to compress data by representing repeated patterns in data using pairs which are each composed of a code and a symbol. A code is an unsigned 16-bit integer and a symbol is an 8-bit ASCII character. The decompression algorithm initializes a dictionary containing only the empty word. The key and value for the decompressing dictionary is swapped; each key is instead a code and each value a word.

Trie ADT is to store common prefixes and has a fixed length of 256 because of the ASCII table. Word ADT is to serve as an array to look up which word to write out when decompressing.

Lastly i/o helps to perform efficient file inputting and outputting

Structure Pseudo-code:

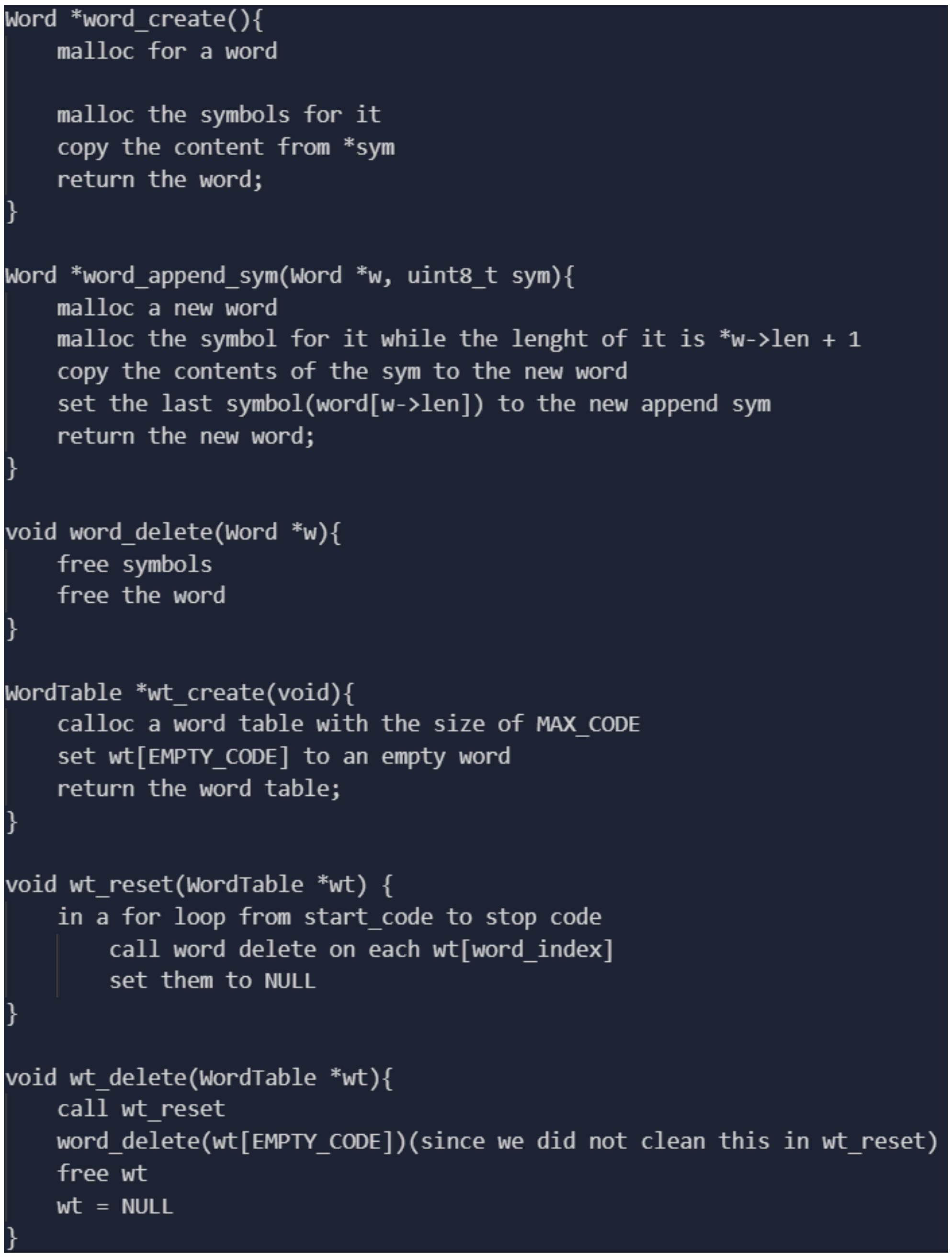
Tries:



```
TrieNode *trie_node_create(uint16_t code){
    malloc for trienode *n
    set each node to NULL
    node's code = code
}

void trie_node_delete(TrieNode *n){
    free(n)
}

TrieNode *trie_create(void){
    create a trie by calling trie_node_create(EMPTY_CODE)
    check if the pointer is NULL or not
    return if success
}

void trie_reset(TrieNode *root){
    loop through the tree since the code is finite(256)
    call node delete on each children and set them to NULL
}

void trie_delete(TrieNode *n){
    for each children on N
        recursively call trie_delete(c)
        set children to NULL
        if we have finished deleting the for loop will break itself
    call trie_node_delete(n)
}

TrieNode *trie_step(TrieNode *n, uint8_t sym){
    return n.children[sym]
}
```

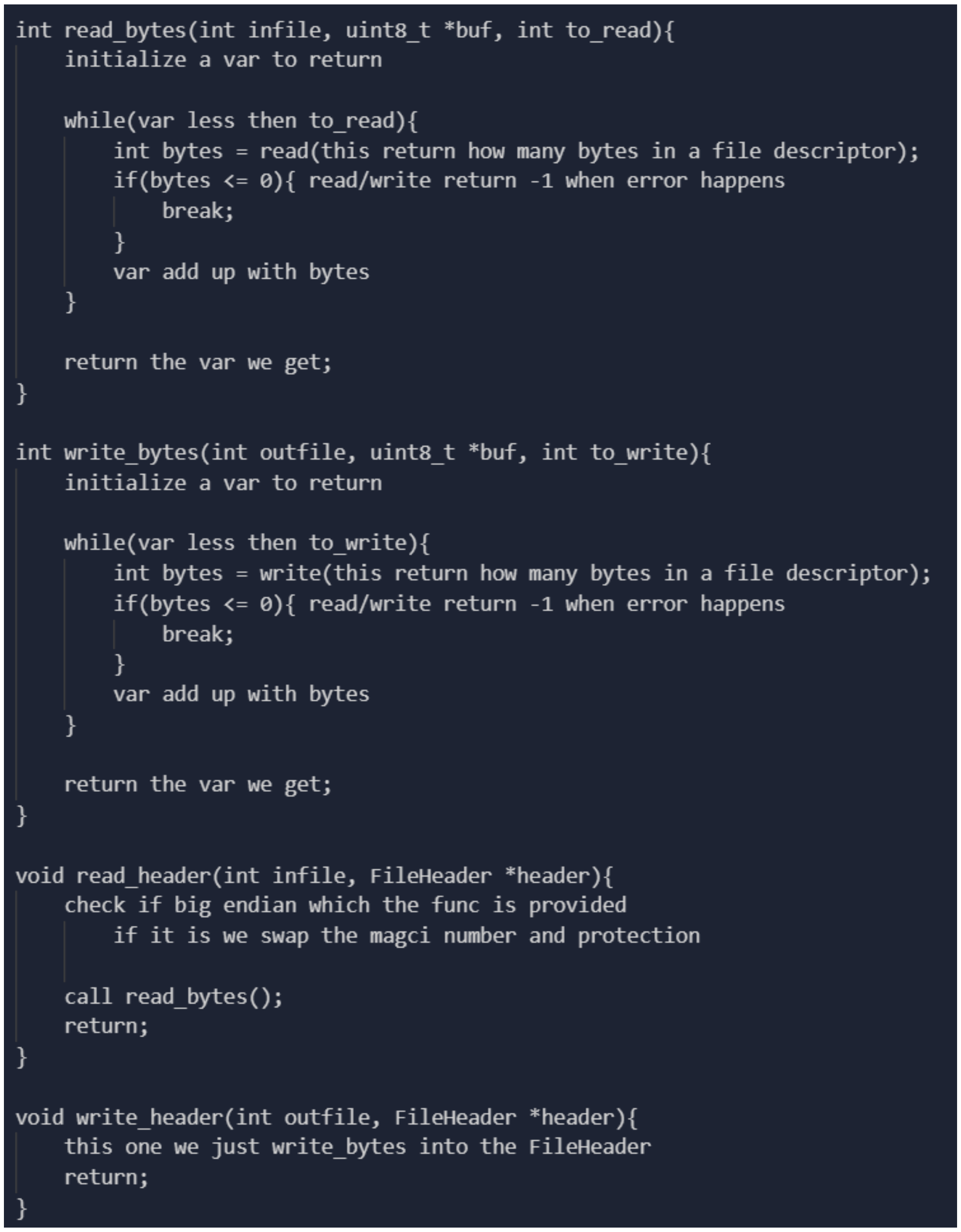
Tries node is used to store characters from words. The code field stores the 16-bit code for the word that ends with the trie node containing the code. This ADT will help reduce the time spent on searching for texts.

Word / WordTable:

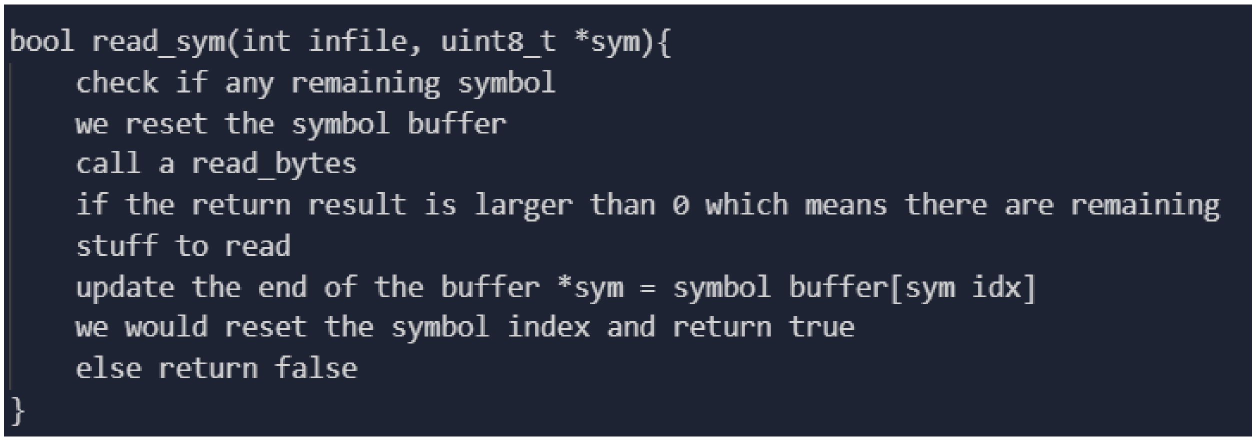


```
Word *word_create(){
    malloc for a word

    malloc the symbols for it
    copy the content from *sym
    return the word;
}

Word *word_append_sym(Word *w, uint8_t sym){
    malloc a new word
    malloc the symbol for it while the lenght of it is *w->len + 1
    copy the contents of the sym to the new word
    set the last symbol(word[w->len]) to the new append sym
    return the new word;
}

void word_delete(Word *w){
    free symbols
    free the word
}

WordTable *wt_create(void){
    calloc a word table with the size of MAX_CODE
    set wt[EMPTY_CODE] to an empty word
    return the word table;
}

void wt_reset(WordTable *wt) {
    in a for loop from start_code to stop code
        call word delete on each wt[word_index]
        set them to NULL
}

void wt_delete(WordTable *wt){
    call wt_reset
    word_delete(wt[EMPTY_CODE])(since we did not clean this in wt_reset)
    free wt
    wt = NULL
}
```

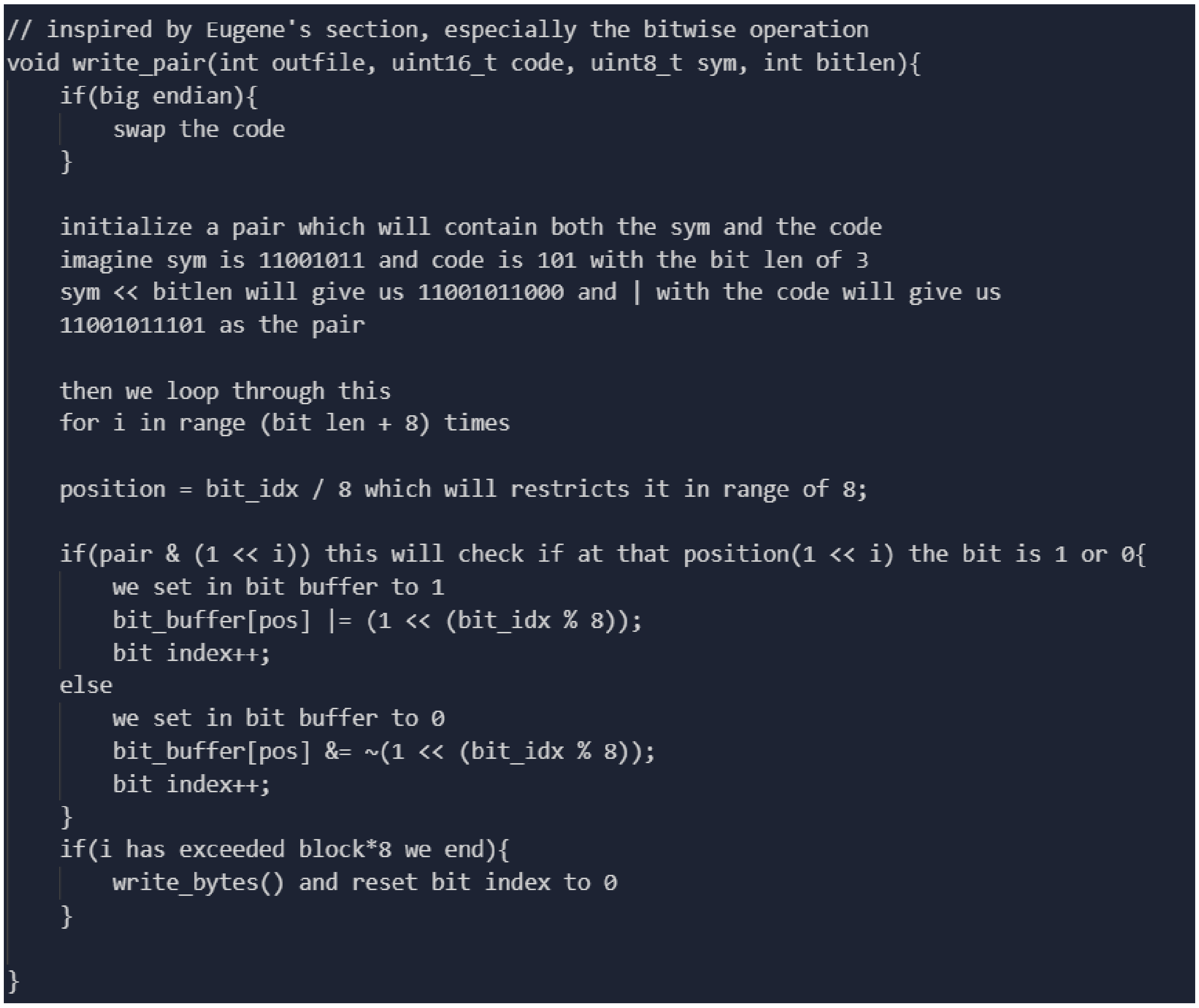
I/O:



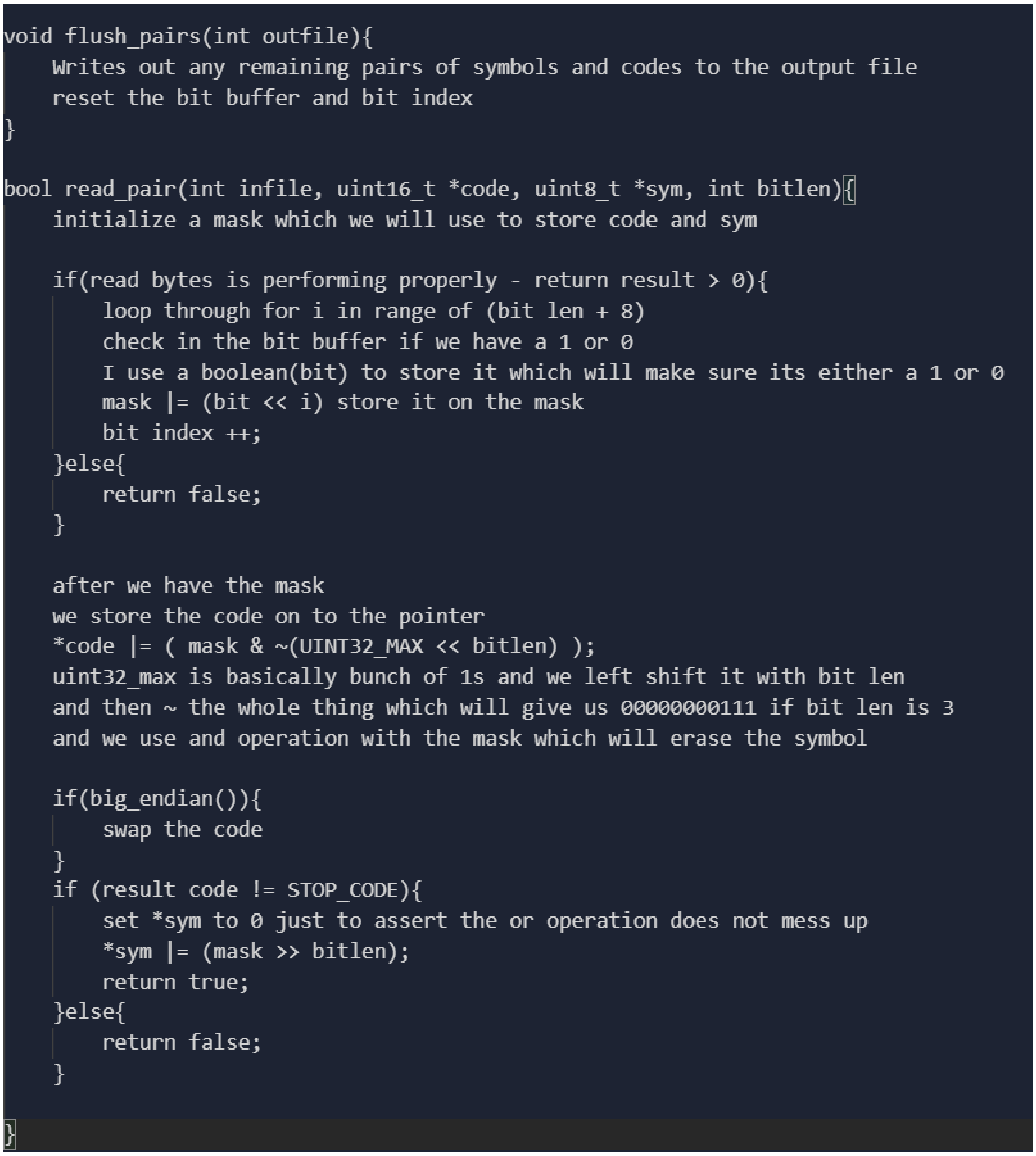
```
int read_bytes(int infile, uint8_t *buf, int to_read){
    initialize a var to return

    while(var less then to_read){
        int bytes = read(this return how many bytes in a file descriptor);
        if(bytes <= 0){ read/write return -1 when error happens
            break;
        }
        var add up with bytes
    }

    return the var we get;
}

int write_bytes(int outfile, uint8_t *buf, int to_write){
    initialize a var to return

    while(var less then to_write){
        int bytes = write(this return how many bytes in a file descriptor);
        if(bytes <= 0){ read/write return -1 when error happens
            break;
        }
        var add up with bytes
    }

    return the var we get;
}

void read_header(int infile, FileHeader *header){
    check if big endian which the func is provided
        if it is we swap the magci number and protection

    call read_bytes();
    return;
}

void write_header(int outfile, FileHeader *header){
    this one we just write_bytes into the FileHeader
    return;
}
```



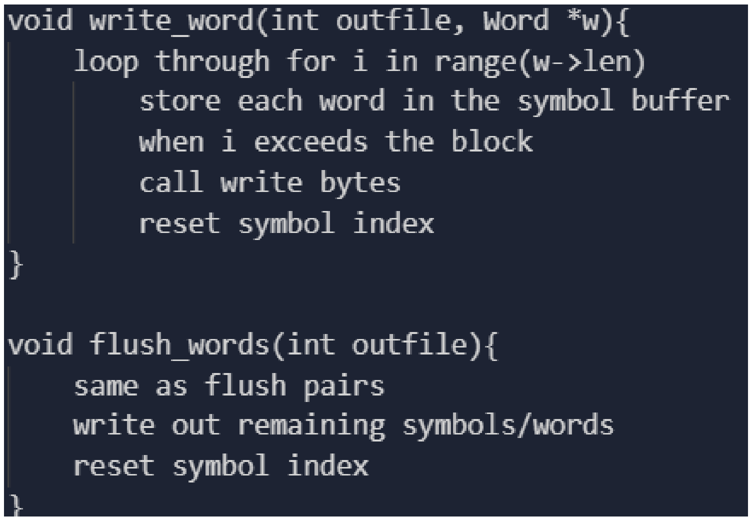
```
bool read_sym(int infile, uint8_t *sym){
    check if any remaining symbol
    we reset the symbol buffer
    call a read_bytes
    if the return result is larger than 0 which means there are remaining
    stuff to read
    update the end of the buffer *sym = symbol buffer[sym idx]
    we would reset the symbol index and return true
    else return false
}
```



```
// inspired by Eugene's section, especially the bitwise operation
void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen){
    if(big endian){
        swap the code
    }

    initialize a pair which will contain both the sym and the code
    imagine sym is 11001011 and code is 101 with the bit len of 3
    sym << bitlen will give us 11001011000 and | with the code will give us
    11001011101 as the pair

    then we loop through this
    for i in range (bit len + 8) times

    position = bit_idx / 8 which will restricts it in range of 8;

    if(pair & (1 << i)) this will check if at that position(1 << i) the bit is 1 or 0{
        we set in bit buffer to 1
        bit_buffer[pos] |= (1 << (bit_idx % 8));
        bit index++;
    else
        we set in bit buffer to 0
        bit_buffer[pos] &= ~(1 << (bit_idx % 8));
        bit index++;
    }
    if(i has exceeded block*8 we end){
        write_bytes() and reset bit index to 0
    }

}
```



```
void flush_pairs(int outfile){
    Writes out any remaining pairs of symbols and codes to the output file
    reset the bit buffer and bit index
}

bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen){
    initialize a mask which we will use to store code and sym

    if(read bytes is performing properly - return result > 0){
        loop through for i in range of (bit len + 8)
        check in the bit buffer if we have a 1 or 0
        I use a boolean(bit) to store it which will make sure its either a 1 or 0
        mask |= (bit << i) store it on the mask
        bit index ++;
    }else{
        return false;
    }

    after we have the mask
    we store the code on to the pointer
    *code |= ( mask & ~(UINT32_MAX << bitlen) );
    uint32_max is basically bunch of 1s and we left shift it with bit len
    and then ~ the whole thing which will give us 00000000111 if bit len is 3
    and we use and operation with the mask which will erase the symbol

    if(big_endian()){
        swap the code
    }
    if (result code != STOP_CODE){
        set *sym to 0 just to assert the or operation does not mess up
        *sym |= (mask >> bitlen);
        return true;
    }else{
        return false;
    }

}
```

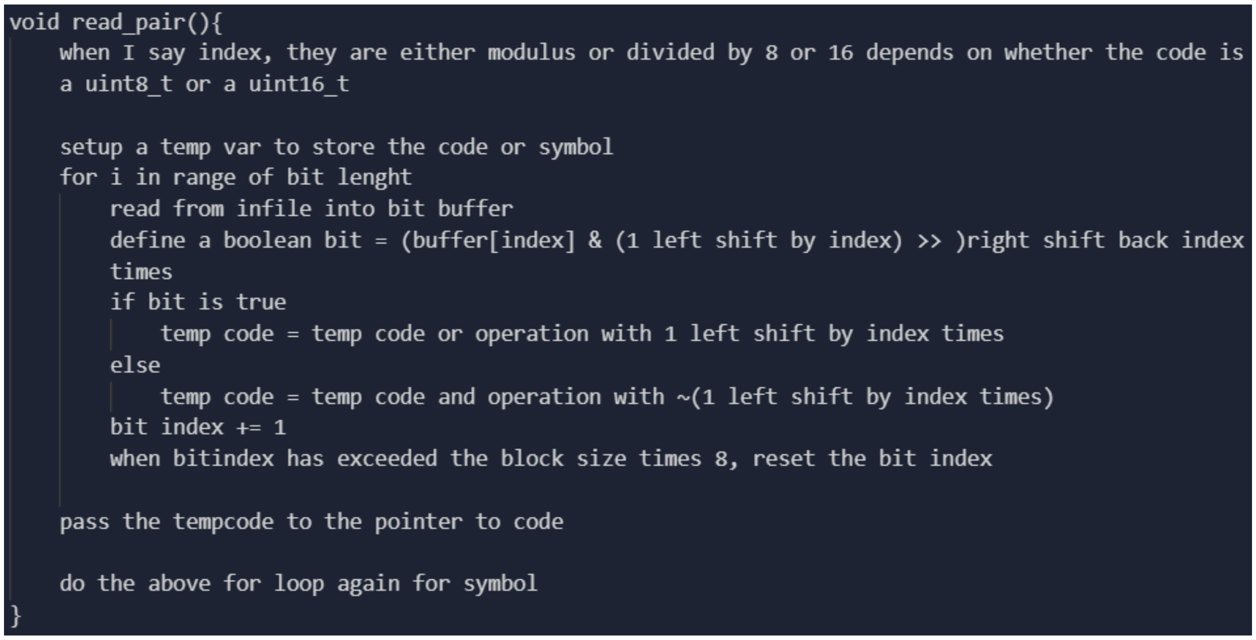


```
void write_word(int outfile, Word *w){
    loop through for i in range(w->len)
        store each word in the symbol buffer
        when i exceeds the block
        call write bytes
        reset symbol index
}

void flush_words(int outfile){
    same as flush pairs
    write out remaining symbols/words
    reset symbol index
}
```

i/o can be lengthy and complicated at first but going over each function while designing helps me order the whole process of encoding and decoding, all pseudo instructions are explained with these screenshots above.

Encode/Decode:
All the pseudocodes are already given in the pdf for them.

Design V2.0:
I tried to do read/write pair by putting code and symbol into one pair(uint64_t) in the above pseudo code and do them with one for loop(i = 0; i < bitlen + 8)
However, that gives me either a seg or wrong error stats(With my understanding I think, it's because I'm exceeding the size of the bit buffer If I tried to do symbol and code together, I had to separate them.)
Thus, I decided to follow Eugene's words and change my read/pair into



```
void read_pair(){
    when I say index, they are either modulus or divided by 8 or 16 depends on whether the code is
    a uint8_t or a uint16_t

    setup a temp var to store the code or symbol
    for i in range of bit lenght
        read from infile into bit buffer
        define a boolean bit = (buffer[index] & (1 left shift by index) >> )right shift back index
        times
        if bit is true
            temp code = temp code or operation with 1 left shift by index times
        else
            temp code = temp code and operation with ~(1 left shift by index times)
        bit index += 1
        when bitindex has exceeded the block size times 8, reset the bit index

    pass the tempcode to the pointer to code

    do the above for loop again for symbol
}
```

write pair is similar to this.(besides differences in bitwise operations)

Summary:
This lab can be complex at first, but after solving/coding them step by step, I get to know more about how we were able to program lossless file compression. Nonetheless, it’s still painful when debugging since so many scenarios, test files, and edge cases can ruin your whole program.