Artyom Martirosyan
11/27/2019
Assignment 8 Design

Assignment Description:

For this assignment we are creating a program which will compress and decompress files.

V.0

Trie functions:

Note you will need to make appropriate headers as well

Inside of the header you will also need to make a Trienode struct which will hold two things: TrieNode *children[256](potential branching); uint16_t code(code of word);

```
TrieNode *trie_node_create(uint16_t code) {
    TrieNode *N = (TrieNode*)calloc( 1,sizeof(TrieNode));
    n->code = code;
    Return n
}
void trie_node_delete(TrieNode *n) {
    free(n)
    N = null
}

TrieNode *trie_create(void) { //creating the root
    TrieNode *n = *trie_node_create(1)
    for(int i =0; i <256, i++) {
        n->children[i] =(char) i;
    }
    Return n;
}
void trie_reset(TrieNode *root) {
    TrieNode temp = root
    trie_delete(root)
    *root = temp;

}
void trie_delete(TrieNode *n) {
```

```
        for(uint32 i = 0; i <256; 1++) {
                if(n->children[i]) {
                        trie_delete(n->children[i]);
                }
        }
        trie_node_delete(n);
}
TrieNode *trie_step(TrieNode *n, uint8_t sym){
                Return n->children[sym]
}
```

IO structs:

Magic: Magic number for compression file_size: size of file Protection: holds the original mask, padding: for padding the bits in order to align the struct

Static uint8_t buffer[4096]

Int buffercounter = 0;

IO functions:

```
void read_header(int infile , FileHeader *header) {
        read(infile, header, sizeof(FileHeader *))
        if(header->magic != MAGIC) {
                printf("wrong magic");
                exit(-1)
        }
}
void write_header(int outfile , FileHeader *header) {
        write(outfile, header, sizeof(header));
}
uint8_t next_char(int infile) {
        if(buffercounter ==0) {
        read(infile, buffer, 4096)
        }
        Buffercounter += 1
        return(charbuffer[buffercounter-1]);
}
void buffer_code(int outfile, uint16_t code, uint8_t bit_len){
        For (int i = 0; i < bit_len; i++) {
                Buffer[buffercounter++] = code[i]
```

```
        }
        Buffercounter = buffercounter+bit_len
}
void flush_codes(int outfile) {
        uint32_t byte_length;
          if (buffercounter % 8 != 0) {
                  byte_length = (buffercounter / 8) + 1;
         } else {
          byte_length = buffercounter / 8;
        }
        write(outfile, buffer, byte_length )


}
uint16_t next_code(int infile , uint8_t bit_len) {
        Uint16_t sum = 0;
        Uint16_t binary = 1;
        If (binarybuffercounter = 4095) {
                flush_codes(infile)
                Binarybuffercounter = 0;

        }
        If (binarybuffercounter==0) {
        read(infile, bitbuffer, 4096)
        }
        for(int i = 0; i < bit_len; i++) {
                if(bitbuffer[binarybuffercounter + i]) {
                Sum = sum +binary
                }
                Binary = binary * 2;
        }
        Binarybuffercounter = binarybuffercounter + bit_len
        Return sum
}
void buffer_word(int outfile , Word *w) {
        for (int i = 0; i < w->word_len; 1++) {
                //might need to check if buffercount is full
                Charbuffer[charbuffercount + i] = w->word[i]
```

```
        }
        Charbuffercount = charbuffercount + w->word_len
}
void flush_words(int outfile) {
uint32_t byte_length;
          if (buffercounter % 8 != 0) {
                    byte_length = (buffercounter / 8) + 1;
          } else {
           byte_length = buffercounter / 8;
          }
          write(outfile, charbuffer, byte_length )
}
```

Word structs:

uint8_t *word: array to hold the word
uint64_t length: length of the word

Word functions:

```
Word *word_create(uint8_t *word, uint64_t length) {
        Word *w = (Word *)malloc(sizeof(Word))
        w->length = length
        w->word[length];
        for(int i = 0; i < length, i++) {
        w->word[i] = word[i]
        }
        Return w
}
void word_delete(Word *w) {
        free(w)
        w->null
        return
}
WordTable *wt_create(void) {
        WordTable *wt = (WordTable *)malloc(sizeof(WordTable))
        wt->entries = (uint8_t *)calloc(65536, sizeof(word*))
        for(uint32_t i = 0; i >256; i++ ) {
                wt->entries[i] = word_create(i, 1);
        }
```

```
        Return wt
}
void wt_delete(WordTable *wt) {
        for(uint i = 0; i < 65536 ) {
                if(wt->entries[i] != 0) {
                        word_delete(wt->entries[i]);
                }
        }
        free(wt->entries)
        wt->entries = NULL
        free(wt)
        wt = NULL
}
void wt_reset(WordTable *wt) {
        WordTable temp = wt
        wt_delete(wt)
        wt = temp
}
void word_print(Word *w) {
        printf("%c", w);
        return
}
void wt_print(WordTable *wt) {
        for(int i = 0; i < 65536; i++) {
                If (wt->entries[i] != 0) {
                        word_print(wt->entries[i]);
                }
        }
}
Main.c {
Include all headers as well as standard libraries
I.e sys/types.h, sys/stat.h, fcntl.h, unistd.h
        Int fileinput = 0;
        Int fileoutput = 0;
        while(getopt options (-vcdi:o:)) {
                If v {verbose = true}
                If c {compression = true }
```

```
            If d {decompression = true}
            If i {char * input = strdup(optarg)}
            If o {char* output = strdup(optarg)}
        }
        if(compression) {
            Follow psuedo code given by eugene
        }

        if(decompression) {
            Follow psuedo code given by eugene
        }
}
```

V2.0

Trie functions:

Note you will need to make appropriate headers as well

Inside of the header you will also need to make a Trienode struct which will hold two things: TrieNode *children[256](potential branching); uint16_t code(code of word);

```
TrieNode *trie_node_create(uint16_t code) {
        TrieNode *N = (TrieNode*)calloc( 1,sizeof(TrieNode));
        n->code = code;
        Return n
}
void trie_node_delete(TrieNode *n) {
        free(n)
        N = null
}

TrieNode *trie_create(void) { //creating the root
        TrieNode *n = *trie_node_create(1)
        for(from 0-256) {
            n->children[i] =make new node(i)
        }
        Return n;
}
void trie_reset(TrieNode *root) {
        Make a loop form zero - 256
```

```
        Do trie delete on child
        Re-add the child after your done deleting it

}
void trie_delete(TrieNode *n) {
        for(fomr 0-256) {
                if(child exists) {
                        trie_delete(n->children[i]);
                }
        }
        trie_node_delete(n);
}
TrieNode *trie_step(TrieNode *n, uint8_t sym){
                Return n->children[sym]
}
```

IO structs:

Magic: Magic number for compression file_size: size of file Protection: holds the original mask, padding: for padding the bits in order to align the struct

```
Static uint8_t bitbuffer[4096]
Int bitbuffercounter = 0;
Static uint8_t chabuffer[4096]
Int charbuffercounter = 0
```

IO functions:

```
void read_header(int infile , FileHeader *header) {
        Read the header
        if(if the magic is wrong) {
                printf(“wrong magic”);
                exit(-1)
        }
}
void write_header(int outfile , FileHeader *header) {
        write(header);
}
uint8_t next_char(int infile) {
If the char counter is full do a flush
Incriment counter
```

```
Return char_buffer[counter-1]
}
void buffer_code(int outfile, uint16_t code, uint8_t bit_len){
        for (0-bit_len){
                If counter is full do a flush
                if(get bit on code) {
                        Set bit(using counter)
                } else {
                        Clear bit(using counter)
                }
                Incriment index
        }
}
void flush_codes(int outfile) {
        uint32_t byte_length;
          if (buffercounter % 8 != 0) {
                   byte_length = (buffercounter / 8) + 1;
         } else {
          byte_length = buffercounter / 8;
        }
        write(outfile, buffer, byte_length )


}
uint16_t next_code(int infile , uint8_t bit_len) {
        Int binary = 1
        Int sum = o
        for(0-bit_len) {
                If empty or full do a read
                if(get bit using bookmark == 1) {
                Add binary to sum
                }
                Multply binary by 2
                Increment counter
        }
}
void buffer_word(int outfile , Word *w) {
        for (int i = 0; i < w->word_len; 1++) {
```

```
        If full do a flush
        Copy over word into buffer
        Increment buffercounter
}
void flush_words(int outfile) {
write(buffer)
Counter = zero
}
```

Word structs:

uint8_t *word: array to hold the word
uint64_t length: length of the word

Word functions:

```
Word *word_create(uint8_t *word, uint64_t length) {
        Word *w = (Word *)malloc(sizeof(Word))
        w->length = length
        w->word= malloc...;
        for(0-length) {
        w->word[i] = word[i]
        }
        Return w
}
void word_delete(Word *w) {
        free(w->word)
        free(w)
        w->null
        return
}
WordTable *wt_create(void) {
        WordTable *wt = (WordTable *)malloc(sizeof(WordTable))
        wt->entries = (uint8_t *)calloc(65536, sizeof(word*))
        for(from 0-255 ) {
                wt->entries[i] = word_create(i, 1);
        }
        Return wt
}
void wt_delete(WordTable *wt) {
```

```
        for(i-65536 ) {
                word_delete(wt->entries[i]);
        }
        Free wt
}
void wt_reset(WordTable *wt) {
        for(256-uint16max) {
                Set that entry to null
        }
}
Main.c {
Include all headers as well as standard libraries
I.e sys/types.h, sys/stat.h, fcntl.h, unistd.h
        Int fileinput = 0;
        Int fileoutput = 0;
        while(getopt options (-vcdi:o:)) {
                If v {verbose = true}
                If c {compression = true }
                If d {decompression = true}
                If i {char * input = strdup(optarg)}
                If o {char* output = strdup(optarg)}
        }
        if(compression) {
                Follow psuedo code given by eugene
                bit_len = log2(next_avail_code) + 1
                buffer_code(curr_node.code ,bit_len)
                scurr_node.children[curr_char]=trie_node_create(next_avail_code)
                curr_node = root.children[curr_char]
                next_avail_code += 1
                encoded_chars += 1
                if (next_avail_code == UINT16_MAX):
                trie_reset ()
                curr_node = root.children[curr_char]
                next_avail_code = 256
                bit_len=log2(next_avail_code)+127buffer_code(curr_code,bit_len)28flush_
        codes ()
        }
```

```
if(decompression) {
        Follow psuedo code given by eugene
        read_header ()
        table = wt_create ()
        next_avail_code = 256
        reset = false
        while (decoded_chars != header.file_size)
        bit_len = log2(next_avail_code + 1) + 1
        curr_code = next_code(bit_len)
        curr_entry = table[curr_code]
        if (decoded_chars == 0 or reset)
        buffer_word(curr_entry)
        prev_word = curr_char
        reset = false
        elif (curr_entry != NULL)
        curr_word = curr_entry.word
        prev_entry = table[prev_code]
        prev_word = prev_entry.word
        new_word = prev_word.append(curr_word [0])
        table[next_avail_code] = word_create(new_word)
        next_avail_code += 1
        buffer_word(curr_entry)
        decoded_chars += curr_entry.length
        Else:
        prev_entry = table[prev_code]
        prev_word = prev_entry.word
        curr_word = prev_word.append(prev_word [0])
        missing_entry = word_create(curr_word)
        table[next_avail_code] = missing_entry
        next_avail_code += 1
        buffer_word(missing_entry)
        decoded_chars += missing_entry.length
        prev_code = curr_code
        if (next_avail_code == UINT16_MAX - 1)
        wt_reset ()38next_avail_code = 256
        reset = true
        flush_words (
}
```
If both or neither are called print something and exit

```
        Run verbose option
}
```