

CSE13S Fall 2020
Assignment 7
Design Document

Goal

The goal for this assignment is to learn about Lempel-ziv compression and decompression. In this assignment, we will be using a Tries ADT in order to store words we can compress. We will also be using a Word ADT to decompress 'codes'. We will also be collecting stats about the files we encode and decode.

File I have to implement + their goal

- Encode: compresses files, texts, and binaries
- Decode: decompresses files, texts, and binaries
- Trie: the ADT for TrieNode
- Word: the ADT for Word
- Io.c: source file for i/o implementation

Trie

- TrieNode *trie_node_create(uint16_t code)
 - Allocate memory for TrieNode* using malloc
 - Set the n->code to code
 - If allocating was successful
 - For i in range (1, alphabet (or 256):
 - trienode->children[i] = NULL
 - Return trie node
- void trie_node_delete(TrieNode *n)
 - free(n)
 - n = null
- TrieNode *trie_create(void)
 - Returns an empty code
 - Return trie_code_create(empty_code)
- void trie_reset(TrieNode *root)
 - For i in alphabet
 - If child at index i is not null
 - trie_delete the child at index i
 - Set children index i to NULL
- void trie_delete(TrieNode *n)
 - Delete children and root recursively
 - For i in alphabet
 - trie_delete(children[i])
 - Set child to null
 - trie_node_delete(n)

- `TrieNode *trie_step(TrieNode *n, uint8_t sym)`
 - Checks the node to see if the sym is part of it
 - If `n->child[sym]` is null
 - Return null
 - Else return `n->child[sym]`

Word

- `Word *word_create(uint8_t *syms, uint32_t len)`
 - `Word *w` = allocate memory using `calloc` and size of words
 - Set `w->len` to `len`
 - `w->syms` = use `calloc` to allocate memory of the sym
 - Copy `w->syms` into `w->syms` either using `memcpy` or a loop
 - Depends on what works with my code !)
- `Word *word_append_sym(Word *w, uint8_t sym)`
 - Creates a new word so we can append sym to it
 - `Word *nw` = allocate memory using `calloc` and size of words
 - `nw->len` = `w->len + 1`
 - `nw->syms` = `(uint8_t *)calloc(len, sizeof(uint8_t))`
 - Copy `w->syms` into `nw->syms`
 - `nw->syms[w->len] = sym`
 - Return `nw`
- `void word_delete(Word *w)`
 - Free `w`
 - Set `w = null`
- `WordTable *wt_create(void)`
 - `WordTable *wt` = allocate memory for is using size of word
 - `wt[empty_code]` = allocate memory using size word
 - Return `wt`
- `void wt_reset(WordTable *wt)`
 - For `i` in range `(1, len)`
 - Delete `wt->syms[i]`
 - Set it to null

IO

- `int read_bytes(int infile, uint8_t *buf, int to_read)`
 - Loop until there are no bytes left && we have read all the bytes in `to_read`
 - Currently reading = `read(infile, buf + total read, to_read - total read)`
 - Return num of bytes read aka counter
- `int write_bytes(int outfile, uint8_t *buf, int to_write)`
 - Loop until there are no bytes left && we have written all the bytes in `to_write`
 - Currently reading = `read(infile, buf + total written, to_read - total write)`
 - Return num of bytes written aka counter
- `void read_header(int infile, FileHeader *header)`
 - `read_byte(infile, header, sizeof(FileHeader))`

- If !small_endian
 - Swap endianness of both magic and protection
- void write_header(int outfile, FileHeader *header)
 - write_byte(infile, header, sizeof(FileHeader))
 - If !small_endian
 - Swap endianness of both magic and protection
- bool read_sym(int infile, uint8_t *sym)
 - Counter to check if we are at the end of a block = -1
 - If the index for the sym buff is empty
 - Read a byte: using read_bytes(infile, symbuff, sym index)
 - Counter to check if we are at the end of a block += bytes + 1
 - Since we read a byte
 - Pass back the buffer at the specified index
- void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)
 - Loop until we read the bitlen
 - Get the bit of code at index of i
 - If the bit == 1;
 - Set the bit
 - Else
 - Clear the bit
 - Increment our buff index
 - If the index is at the end of block * 8
 - Call write bytes
 - Reset the index
 - Repeat again for sym
- void flush_pairs(int outfile)
 - write_bytes(outfile, bitbuff, bytes of bit index);
- bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)
 - Zero out code
 - Zero out sym
 - Loop until we read the bitlen
 - Get the bit of code at index of i
 - If the bit == 1;
 - Set the bit
 - Else
 - Clear the bit
 - Increment our buff index
 - If the index is at the end of block * 8
 - Call write bytes
 - Reset the index
 - Repeat again for sym
 - If *code is == stop code return false
- void write_word(int outfile, Word *w)
 - For i in range(0, len of word)

- Copy symbuff [index of symbuff] to sym[i]
 - Increment sym index
- If sym index == end of of block
 - Write bytes
- void flush pairs (int outfile)
 - write _bytes (outfile, sym buff, bytes of sym index);

Compression

- Take command line options: v, i, o
- Open file
- If i wasn't specified, i = stdin
- Use fstat
- Open outfile
- If i wasn't specified, o = stdout
- Write out the filled file header
- Create a tie
 - See the pseudo for the specifics things each thing ==
- Init a counter to keep track of next available word
- Create counters prev node and prev sysm
- Use read_sysm in a loop until it returns false
 - See pseudo for specifics things needed to do inside the loop
- Check of current node == root trie node
 - Increment counter
- Write the pair
- Flush pair
- Close the files
- Free the memory of trie node

Decompression

- Take command line options: v, i, o
- Open file
- If i wasn't specified, i = stdin
- Use fstat
- Open outfile
- If o wasn't specified, o = stdout
- Write out the filled file header
- Create a word table
 - See the pseudo for the specifics things each thing ==
- Init a counter to keep track of current and next code
- Use read_apir in a loop until it returns false
 - See pseudo for specifics things needed to do inside the loop
- Check of code == stop_code
 - Increment counter
- Flush pair
- Close the files
- Free the memory of word table

Updates

This lab was definitely one of the hardest labs I've ever had to do. While coding this lab, I ran into lots of trouble with bit manipulation. Some of the hardest aspects of the lab included understanding how everything worked together and really understanding bit buffering. Although I spent over 80 on this lab and tried my best, I was not able to get either one of my encodes and decode to work. I did however, learn a lot about trie nodes and words ADT. During the lab, I tried to code function by function and testing two functions at a time to see if they worked, but even after that debugging my errors were pretty hard since they could be coming from anywhere in my functions. Although I didn't complete the lab, I actually had fun doing it and appreciated all the different elements, such as file permissions, I was able to learn.