Assignment 6 Design

Lempel-Ziv Compression

Arnav Nepal

March 1, 2023

1 Introduction

In this assignment we will be implementing the 1978 Lempel-Ziv data compression algorithm, otherwise known as LZ78. LZ78 uses a dictionary to map keywords to a "code", or index. Keywords are first input into the dictionary as single characters. When a repeat character in a text file is encountered, the dictionary stores is as a pair of the index of the repeated character and the new character. It repeats this, performing the same action any time it runs into a repeated character or sequence of characters. In order to make our program we will need to design and implements a host of abstract data types (ADT's), including a trie ADT and word table ADT. The files we will be implementing include:

- encode.c
- decode.c
- trie.c
- word.c
- io.c
- Makefile

2 Design and Psuedocode

This program requires us to implement 3 abstract data types: a trie module, a word table module, and an input/output module.

2.1 trie.c

A trie, a play on "re*trie*val", is a tree data structure. Also known as a prefix tree, each node in a trie contains a symbol and a pointer to n child nodes, where n is the number of letters in the alphabet being used. As we are using ASCII, $n = 256$. We will be using the trie data structure during compression to store words.

```
//reference for TrieNode struct
struct TrieNode {
    TrieNode *children[ALPHABET];
    uint16_t code;
};

//ALPHABET is defined as 256

TrieNode *trie_node_create(uint16_t index):
    allocate memory for trienode struct *node
    set node->code = index
    for i in range (ALPHABET):
        set node->children[i] to NULL
    return node

//destrctor for trie nodes. only a single free() required
//here since struct elements don't need memory here
void trie_node_delete(TrieNode *n);
    free(n)
    set n = NULL

//simple function, just needs to create a node
//and set code to EMPTY_CODE
TrieNode *trie_create(void);
    TrieNode *root = trie_node_create(EMPTY_CODE)
    if root != NULL:
        return root
    else:
        return NULL

//destructor for trie. this function first checks to see if a given node
//holds any child nodes. If it does, it recursively calls iteslf. If no
//child nodes are held, the end of a branch has been reached, and thus it
//is safe to start deleting nodes. This function can clear a trie or any
//subtrie within a trie
void trie_delete(TrieNode *n);
    for i in range(ALPHABET):
        if n->children[i] == NULL:
            continue
        else:
            trie_delete(n->children[i]) //recursive call
    trie_node_delete(n)

//simple function once trie_delete made. calls trie_delete
//on every child node of the root node
```

```
void trie_reset(TrieNode *root):
    for i in range(ALPHABET):
        if root->children[i] == NULL:
            continue
        else:
            trie_delete(root->children[i])


TrieNode *trie_step(TrieNode *n, uint8_t sym);
    for i in range (ALPHABET):
        if struct pointed to in index i holds sym:
            return n->children[i]
    return NULL
```
