Федеральное государственное бюджетное образовательное учреждение

высшего образования «Сибирский государственный университет телекоммуникаций

и информатики»

Факультет ИВТ

Кафедра вычислительных систем

**Курсовая работа**

на тему « СЖАТИЕ ДАННЫХ »

Вариант 4.6 « Алгоритм Лемпела – Зива – Велча, LZW »

Выполнил:

студент гр. ИС-241

Газиев Р.Р.

Проверил:

Фульман В.О.

Новосибирск, 2020

1

**Содержание**

Тема..............................................................................................................................3

Задание

Анализ задачи

Тестовые данные

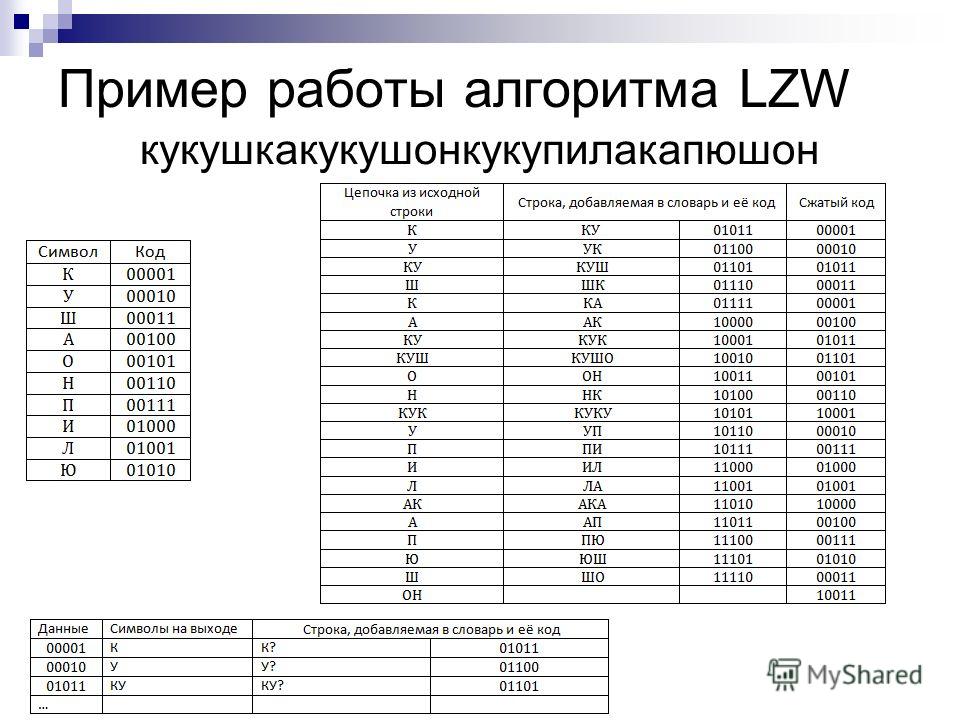
Листинг программы

2

**Тема**

Темой данной курсовой работы является “Сжатие данных” ,а именно использование Алгоритма Лемпела – Зива – Велча или LZW.

Примечательной особенностью алгоритма LZW является простота реализации, благодаря которой он до сих пор очень популярен, несмотря на зачастую худшую степень сжатия по сравнению с такими аналогами, как LZ77. Обычно LZW реализуется с помощью префиксного дерева, содержащего фразы из словаря: для нахождения W нужно просто прочитать как можно более длинную строку из корня дерева, затем для добавления новой фразы WK нужно присоединить к найденному узлу нового сына по символу K, а кодом фразы W может выступать индекс узла в массиве, содержащем все узлы.



3

**Задание**

Реализовать программу lzwcompress сжатия текстовых файлов на английском языке алгоритмом Лемпела – Зива – Велча. Сжатие осуществляется с аргументом командной строки -c (compress), а распаковка – с аргументом -d (decompress). Опция -o указывает имя выходного файла. Вызов программы с аргументами командной стоки может выглядить следующим образом:

$ lzwcompress -c -o file.lzw file.txt # сжатие file.txt в file.lzw

$ lzwcompress -d -o file1.txt file.lzw # распаковка file.lzw в file1.txt

Критерии оценки Оценка «хорошо»: реализован алгоритм сжатия, размер словаря 65536 элементов, при переполнении словаря он полностью сбрасывается (за исключением односимвольных фраз).

Оценка «отлично»: реализован алгоритм сжатия, размер словаря может быть задан пользователем, при переполнении словаря удаляется наименее часто используемая фраза. Дополнительным плюсом будет использование кодов переменной длины (размер ссылки на словарь сначала 1 байт, когда 1 байт переполняется – 2 байта и т.д.). Указание к выполнению задания Алгоритм LZW основан на алгоритме LZ78, он используется в программе сжатия compress ОС Unix, а также в формате GIF. Также как и в LZ78, в алгоритме LZW для декомпрессии не нужно сохранять всю таблицу кодов в файл для распаковки. Алгоритм построен таким образом, что весь словарь можно восстановить, зная все односимвольные фразы и пользуясь потоком кодов.

Основные этапы алгоритма LZW:

**1)**Инициализация словаря всеми возможными односимвольными фразами (обычно 256 символами расширенного ASCII). Инициализация новой фразы ω первым символом сообщения. **2)**Считать очередной символ k из кодируемого сообщения. **3)**Если конец сообщения, записать код для ω в выходной поток, кодирование завершено. **4)**Если фраза ωk уже есть в словаре, присвоить ω = ωk и перейти на шаг 2. Иначе, записать код для ω в выходной поток, добавить ωk в словарь, присвоить ω = k и перейти на шаг 2. Для LZW ключевым для размера получаемых кодов является размер словаря во фразах: LZW-коды имеют постоянную длину, равную округленному в большую сторону двоичному логарифму размера словаря. При переполнении словаря, из него удаляют либо наиболее редко используемую фразу, либо все фразы, отличающиеся от одиночного символа.

4

**Анализ задачи**

1.Педставления данных при кодировании текста алгоритмом LZW:

Данные: На выходе: Новая запись:  
 Полная: Частичная:  
10100 = 20 T 27: T?  
01111 = 15 O 27: TO 28: O?  
00010 = 2 B 28: OB 29: B?  
00101 = 5 E 29: BE 30: E?  
01111 = 15 O 30: EO 31: O?   
10010 = 18 R 31: OR 32: R? <---- начинаем использовать 6- битные группы

001110 = 14 N 32: RN 33: N?  
001111 = 15 O 33: NO 34: O?  
010100 = 20 T 34: OT 35: T?  
011011 = 27 TO 35: TT 36: TO? <---- для 37, добавляем

только первый элемент

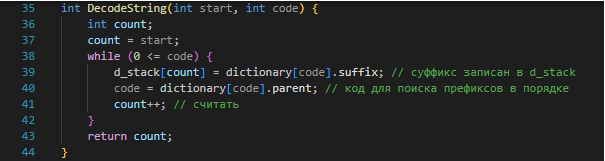
011101 = 29 BE 36: TOB 37: BE? следующего слова

словаря

011111 = 31 OR 37: BEO 38: OR?  
100100 = 36 TOB 38: ORT 39: TOB?  
011110 = 30 EO 39: TOBE 40: EO?  
100000 = 32 RN 40: EOR 41: RN?  
100010 = 34 OT 41: RNO 42: OT?  
000000 = 0 #

2.

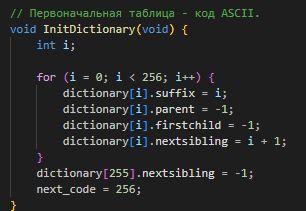
* Строка декодирования.



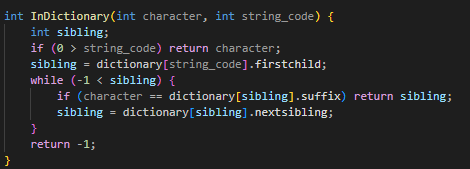
Записав суффикс в стак , в его код записываем префикс для последующего поиска, возвращаем номер нужного нам элемента кода.

5

* Заполнение словаря (первые 256 символов) таблицей ASCII.

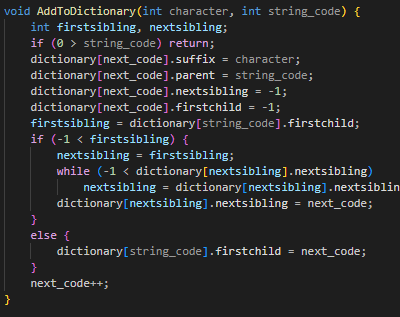


* Словарь инициализации.



Проверяет наличие слова в словаре, если находит возвращает во входной поток.

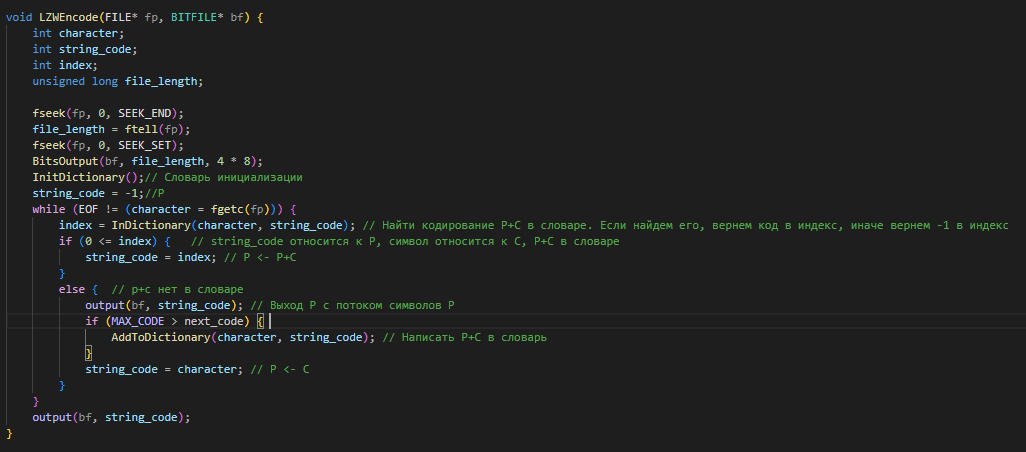
* Добавление в словарь нового элемента языка кодирования



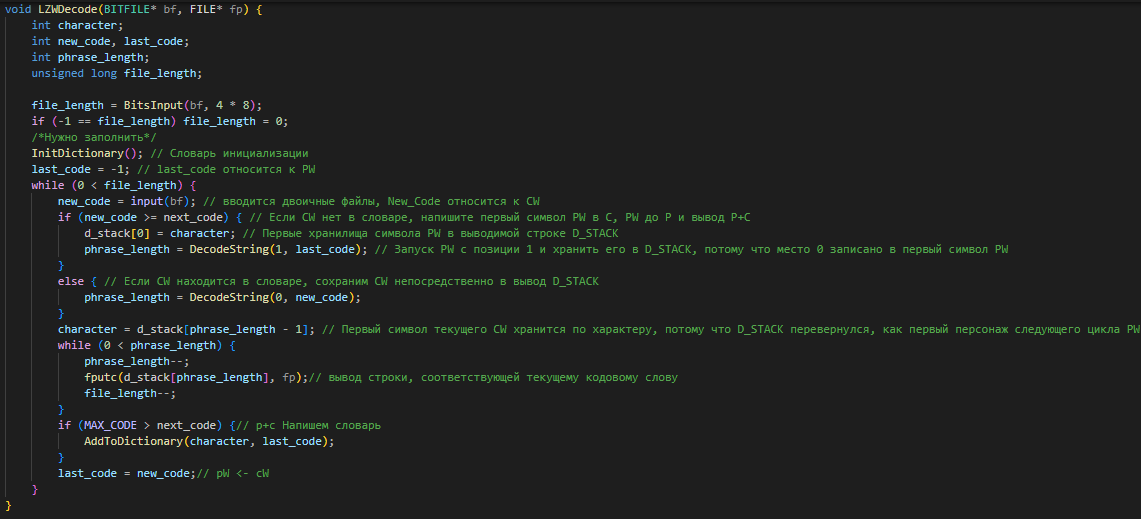
Записывает новый элемент, предварительно проверяя есть ли схожий по написанию.

6

* Функция кодирования входного файла

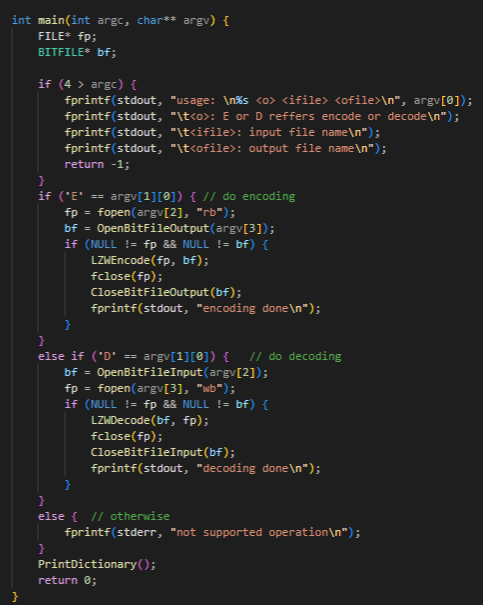


* Функция декодирования в выходной файл



7

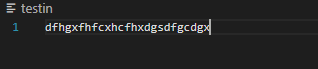
* Основная функция



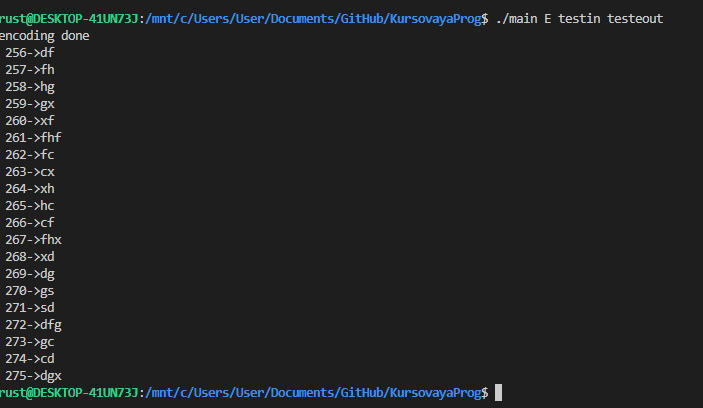
8

**Тестовые данные.**

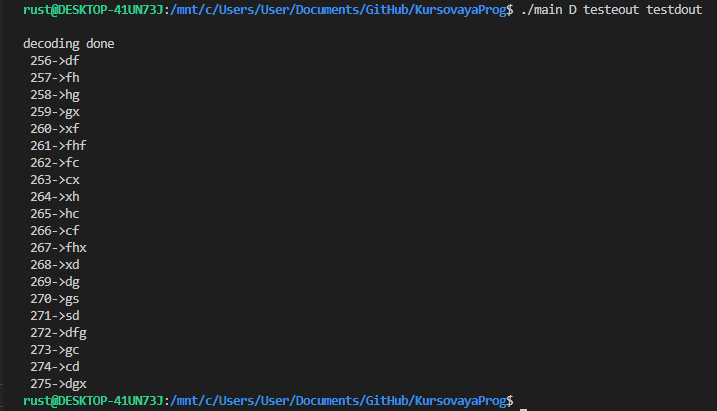
Вводится строка из произвольных символов во входящий файл.



Строка из файла проходит этап кодировки и записывается в словарь.



После происходит декодирование в другой файл .

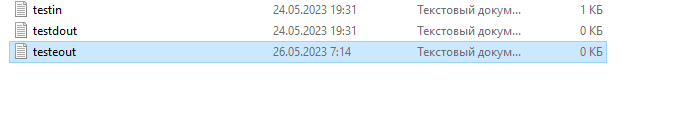


9

Содержание файлов совпадает.



Размер декодируемого файла как и ожидалось оказался меньше кодируемого.



**Листинг программы.**

**main.c**

**#include <stdlib.h>**

**#include <stdio.h>**

**#include "bitio.h"**

**#define MAX\_CODE 65535**

**struct {**

**int suffix;**// суффикс относится к P, родитель относится к C

**int parent, firstchild, nextsibling;** // Последние две переменные используются для выходных символов один за другим

**} dictionary[MAX\_CODE + 1];**

**int next\_code;**// указывает на самое большое кодовое слово в словаре Следующего кода, например, самый большой код в словаре - 255, точка следующего\_кода до 256

**int d\_stack[MAX\_CODE];** // кодовое слово соответствует строке, которая в основном используется для вывода символов в строке в порядке, которая расположена в перевернутом порядке

**#define input(f) ((int)BitsInput( f, 16))**

**#define output(f, x) BitsOutput( f, (unsigned long)(x), 16)**

**int DecodeString(int start, int code);**

**void InitDictionary(void);**

// Печать из 256, кодовое наблюдение, полученное алгоритмом LZW.

**void PrintDictionary(void) {**

**int n;**

**int count;**

**for (n = 256; n < next\_code; n++) {**

**count = DecodeString(0, n);**  10

**printf("%4d->", n);**

**while (0 < count--) printf("%c", (char)(d\_stack[count]));**

**printf("\n");**

**}**

**}**

// Поместите символы в символы, соответствующие строке кода, один за другим в массиве D\_STACK, что удобно для вывода строки

// Когда начало = 0, код = 258,

// Предположим, что кодовое слово 258 соответствует ABA, P = AB, C = A и сначала хранить суффикс, то есть A a in d\_stack [0], затем p = a, c = b и сохранение b в D\_Stack [1], наконец, сохранить A в D\_STACK [2], чтобы облегчить вывод ABA

// Длина строки, соответствующая этому кодовому слову, возвращает

**int DecodeString(int start, int code) {**

**int count;**

**count = start;**

**while (0 <= code) {**

**d\_stack[count] = dictionary[code].suffix;** // суффикс записан в d\_stack

**code = dictionary[code].parent;** // код для поиска префиксов в порядке

**count++;** // считать

**}**

**return count;**

**}**

**void InitDictionary(void) {**

**int i;**

**for (i = 0; i < 256; i++) {**

**dictionary[i].suffix = i;**

**dictionary[i].parent = -1;**

**dictionary[i].firstchild = -1;**

**dictionary[i].nextsibling = i + 1;**

**}**

**dictionary[255].nextsibling = -1;**

**next\_code = 256;**

**}**

**11**

**int InDictionary(int character, int string\_code) {**

**int sibling;**

**if (0 > string\_code) return character;**

**sibling = dictionary[string\_code].firstchild;**

**while (-1 < sibling) {**

**if (character == dictionary[sibling].suffix) return sibling;**

**sibling = dictionary[sibling].nextsibling;**

**}**

**return -1;**

**}**

**void AddToDictionary(int character, int string\_code) {**

**int firstsibling, nextsibling;**

**if (0 > string\_code) return;**

**dictionary[next\_code].suffix = character;**

**dictionary[next\_code].parent = string\_code;**

**dictionary[next\_code].nextsibling = -1;**

**dictionary[next\_code].firstchild = -1;**

**firstsibling = dictionary[string\_code].firstchild;**

**if (-1 < firstsibling) {**

**nextsibling = firstsibling;**

**while (-1 < dictionary[nextsibling].nextsibling)**

**nextsibling = dictionary[nextsibling].nextsibling;**

**dictionary[nextsibling].nextsibling = next\_code;**

**}**

**else {**

**dictionary[string\_code].firstchild = next\_code;**

**}**

**next\_code++;**

**}**

**void LZWEncode(FILE\* fp, BITFILE\* bf) {**

**int character;**

**int string\_code; 12**

**int index;**

**unsigned long file\_length;**

**fseek(fp, 0, SEEK\_END);**

**file\_length = ftell(fp);**

**fseek(fp, 0, SEEK\_SET);**

**BitsOutput(bf, file\_length, 4 \* 8);**

**InitDictionary();**// Словарь инициализации

**string\_code = -1;**//P

**while (EOF != (character = fgetc(fp))) {**

**index = InDictionary(character, string\_code);** // Найти кодирование P+C в словаре. Если найдем его, вернем код в индекс, иначе вернем -1 в индекс

**if (0 <= index) {** // string\_code относится к P, символ относится к C, P+C в словаре

**string\_code = index;** // P <- P+C

**}**

**else {** // p+c нет в словаре

**output(bf, string\_code);** // Выход P с потоком символов P

**if (MAX\_CODE > next\_code) {**

**AddToDictionary(character, string\_code);** // Написать P+C в словарь

**}**

**string\_code = character;** // P <- C

**}**

**}**

**output(bf, string\_code);**

**}**

**void LZWDecode(BITFILE\* bf, FILE\* fp) {**

**int character;**

**int new\_code, last\_code;**

**int phrase\_length;**

**unsigned long file\_length;**

**file\_length = BitsInput(bf, 4 \* 8);**

**if (-1 == file\_length) file\_length = 0;**

**13**

**InitDictionary();** // Словарь инициализации

**last\_code = -1**; // last\_code относится к PW

**while (0 < file\_length) {**

**new\_code = input(bf);** // вводится двоичные файлы, New\_Code относится к CW

**if (new\_code >= next\_code) {** // Если CW нет в словаре, напишите первый символ PW в C, PW до P и вывод P+C

**d\_stack[0] = character;** // Первые хранилища символа PW в выводимой строке D\_STACK

**phrase\_length = DecodeString(1, last\_code);** // Запуск PW с позиции 1 и хранить его в D\_STACK, потому что место 0 записано в первый символ PW

**}**

**else {** // Если CW находится в словаре, сохраним CW непосредственно в вывод D\_STACK

**phrase\_length = DecodeString(0, new\_code);**

**}**

**character = d\_stack[phrase\_length - 1];** // Первый символ текущего CW хранится по character, потому что D\_STACK перевернулся, как первый персонаж следующего цикла PW

**while (0 < phrase\_length) {**

**phrase\_length--;**

**fputc(d\_stack[phrase\_length], fp)**;// вывод строки, соответствующей текущему кодовому слову

**file\_length--;**

**}**

**if (MAX\_CODE > next\_code) {**// p+c Напишем словарь

**AddToDictionary(character, last\_code);**

**}**

**last\_code = new\_code;// pW <- cW**

**}**

**}**

**int main(int argc, char\*\* argv) {**

**FILE\* fp;**

**BITFILE\* bf;**

**if (4 > argc) {**

**fprintf(stdout, "usage: \n%s <o> <ifile> <ofile>\n", argv[0]);**

**fprintf(stdout, "\t<o>: E or D reffers encode or decode\n");**

**fprintf(stdout, "\t<ifile>: input file name\n");**

**fprintf(stdout, "\t<ofile>: output file name\n"); 14**

**return -1;**

**}**

**if ('E' == argv[1][0]) { // do encoding**

**fp = fopen(argv[2], "rb");**

**bf = OpenBitFileOutput(argv[3]);**

**if (NULL != fp && NULL != bf) {**

**LZWEncode(fp, bf);**

**fclose(fp);**

**CloseBitFileOutput(bf);**

**fprintf(stdout, "encoding done\n");**

**}**

**}**

**else if ('D' == argv[1][0]) { // do decoding**

**bf = OpenBitFileInput(argv[2]);**

**fp = fopen(argv[3], "wb");**

**if (NULL != fp && NULL != bf) {**

**LZWDecode(bf, fp);**

**fclose(fp);**

**CloseBitFileInput(bf);**

**fprintf(stdout, "decoding done\n");**

**}**

**}**

**else { // otherwise**

**fprintf(stderr, "not supported operation\n");**

**}**

**PrintDictionary();**

**return 0;**

**}**

**bmain.c**

**#include <stdlib.h>**

**#include <stdio.h>**

**#include "bitio.h" 15**

**BITFILE\* OpenBitFileInput(char\* filename) {**

**BITFILE\* bf;**

**bf = (BITFILE\*)malloc(sizeof(BITFILE));**

**if (NULL == bf) return NULL;**

**if (NULL == filename) bf->fp = stdin;**

**else bf->fp = fopen(filename, "rb");**

**if (NULL == bf->fp) return NULL;**

**bf->mask = 0x80;**

**bf->rack = 0;**

**return bf;**

**}**

**BITFILE\* OpenBitFileOutput(char\* filename) {**

**BITFILE\* bf;**

**bf = (BITFILE\*)malloc(sizeof(BITFILE));**

**if (NULL == bf) return NULL;**

**if (NULL == filename) bf->fp = stdout;**

**else bf->fp = fopen(filename, "wb");**

**if (NULL == bf->fp) return NULL;**

**bf->mask = 0x80;**

**bf->rack = 0;**

**return bf;**

**}**

**void CloseBitFileInput(BITFILE\* bf) {**

**fclose(bf->fp);**

**free(bf);**

**}**

**void CloseBitFileOutput(BITFILE\* bf) {**

**// Output the remaining bits**

**if (0x80 != bf->mask) fputc(bf->rack, bf->fp);**

**fclose(bf->fp);**

**free(bf);**

**} 16**

**int BitInput(BITFILE\* bf) {**

**int value;**

**if (0x80 == bf->mask) {**

**bf->rack = fgetc(bf->fp);**

**if (EOF == bf->rack) {**

**fprintf(stderr, "Read after the end of file reached\n");**

**exit(-1);**

**}**

**}**

**value = bf->mask & bf->rack;**

**bf->mask >>= 1;**

**if (0 == bf->mask) bf->mask = 0x80;**

**return((0 == value) ? 0 : 1);**

**}**

**unsigned long BitsInput(BITFILE\* bf, int count) {**

**unsigned long mask;**

**unsigned long value;**

**mask = 1L << (count - 1);**

**value = 0L;**

**while (0 != mask) {**

**if (1 == BitInput(bf))**

**value |= mask;**

**mask >>= 1;**

**}**

**return value;**

**}**

**void BitOutput(BITFILE\* bf, int bit) {**

**if (0 != bit) bf->rack |= bf->mask;**

**bf->mask >>= 1;**

**if (0 == bf->mask) { // eight bits in rack**

**fputc(bf->rack, bf->fp);**

**bf->rack = 0; 17**

**bf->mask = 0x80;**

**}**

**}**

**void BitsOutput(BITFILE\* bf, unsigned long code, int count) {**

**unsigned long mask;**

**mask = 1L << (count - 1);**

**while (0 != mask) {**

**BitOutput(bf, (int)(0 == (code & mask) ? 0 : 1));**

**mask >>= 1;**

**}**

**}**

**#if 0**

**int main(int argc, char\*\* argv) {**

**BITFILE\* bfi = NULL, \* bfo = NULL;**

**int bit;**

**int count = 0;**

**if (1 < argc) {**

**if (NULL == OpenBitFileInput(bfi, argv[1])) {**

**fprintf(stderr, "fail open the file\n");**

**return -1;**

**}**

**}**

**else {**

**if (NULL == OpenBitFileInput(bfi, NULL)) {**

**fprintf(stderr, "fail open stdin\n");**

**return -2;**

**}**

**}**

**if (2 < argc) {**

**if (NULL == OpenBitFileOutput(bfo, argv[2])) {**

**fprintf(stderr, "fail open file for output\n");**

**return -3; 18**

**}**

**}**

**else {**

**if (NULL == OpenBitFileOutput(bfo, NULL)) {**

**fprintf(stderr, "fail open stdout\n");**

**return -4;**

**}**

**}**

**while (1) {**

**bit = BitInput(bfi);**

**fprintf(stderr, "%d", bit);**

**count++;**

**if (0 == (count & 7))fprintf(stderr, " ");**

**BitOutput(bfo, bit);**

**}**

**return 0;**

**}**

**#endif**

**bitio.h**

**#pragma once**

**#ifndef \_\_BITIO\_\_**

**#define \_\_BITIO\_\_**

**#include <stdio.h>**

**typedef struct {**

**FILE\* fp;**

**unsigned char mask;**

**int rack;**

**}BITFILE;**

**19**

**BITFILE\* OpenBitFileInput(char\* filename);**

**BITFILE\* OpenBitFileOutput(char\* filename);**

**void CloseBitFileInput(BITFILE\* bf);**

**void CloseBitFileOutput(BITFILE\* bf);**

**int BitInput(BITFILE\* bf);**

**unsigned long BitsInput(BITFILE\* bf, int count);**

**void BitOutput(BITFILE\* bf, int bit);**

**void BitsOutput(BITFILE\* bf, unsigned long code, int count);**

**#endif // \_\_BITIO\_\_**

20