WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

Teoria Informacji i Kodowania

Laboratoria nr 3

Osoba realizująca: Mateusz Grabowski	**Prowadzący: Jerzy Dorobisz**
Grupa: WCY21IY4S1	**Data ćwiczenia: 5.06.2023**

Środowisko programistyczne: Microsoft Visual Studio Community 2022 (64-bitowy)
Wersja 17.4.4

Język: C++

Zadanie L3_1

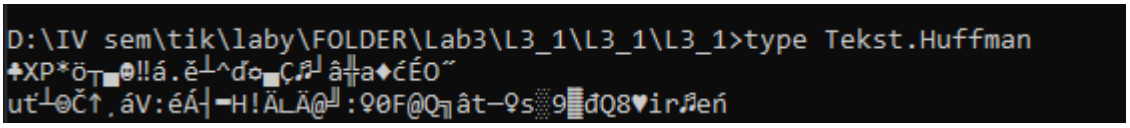
```
ĆWICZENIE LABORATORYNE NUMER 3. ZADANIE 1
KOMPRESJA PLIKU

WYNIK PROGRAMU: PLIK SKOMPRESOWANY
ARGUMENT1: NAZWA PLIKU WEJŚCIOWEGO (nazwa kropka rozszerzenie)
DODATKOWE WYNIKI PROGRAMU:  ZMIENNA GLOBALNA dokumentacja
PLIK WYJŚCIOWY 1. MA TĘ SAMĄ nazwę PLIKU WEJŚCIOWEGO ZE ZMIENIONYM PRZEZ PROGRAM
rozszerzeniem (*.Huffman)
ZAPIS DO PLIKU: BINARNY
Sprawdzenie poprawności wyników za pomoca edytora HxD (sprawdzenie zapisów tekstowych)
pliku *.Huffman
Weryfikacja poprawności kompresji dopiero po dekompresji.
*/
```

```
01111100101000011110011010101
10011111010001010111011010011
Buffer nr.1: (0x81)
Buffer nr.2: (0x87)
Buffer nr.3: (0x41)
Buffer nr.4: (0x0D)
Buffer nr.5: (0x05)
Buffer nr.6: (0x58)
Buffer nr.7: (0x50)
Buffer nr.8: (0x2A)
Buffer nr.9: (0x94)
Buffer nr.10: (0xC2)
Buffer nr.11: (0xDC)
Buffer nr.12: (0x1B)
Buffer nr.13: (0x28)
Buffer nr.14: (0x26)
Buffer nr.15: (0x37)
Buffer nr.16: (0x02)
Buffer nr.17: (0x13)
Buffer nr.18: (0xA0)
Buffer nr.19: (0x2E)
Buffer nr.20: (0xD8)
Buffer nr.21: (0xC1)
Buffer nr.22: (0x5E)
Buffer nr.23: (0xD4)
Buffer nr.24: (0x0F)
Buffer nr.25: (0xDC)
```

```
Buffer nr.4340: (0x84)
Buffer nr.4341: (0x9D)
Buffer nr.4342: (0xA2)
Buffer nr.4343: (0xAD)
Buffer nr.4344: (0x4B)
Buffer nr.4345: (0x52)
Buffer dopelniony: (0xD4)

Rozmiar przed kompresja: 7070
Rozmiar po kompresji: 4347
```

```
Model informacyjny:
Byte:   (0x20), Frequency: 1050
Byte: i (0x69), Frequency: 576
Byte: e (0x65), Frequency: 518
Byte: a (0x61), Frequency: 454
Byte: o (0x6F), Frequency: 354
Byte: n (0x6E), Frequency: 327
Byte: s (0x73), Frequency: 318
Byte: r (0x72), Frequency: 282
Byte: t (0x74), Frequency: 268
Byte: m (0x6D), Frequency: 239
Byte: c (0x63), Frequency: 228
Byte: u (0x75), Frequency: 215
Byte: z (0x7A), Frequency: 200
 (0x0D), Frequency: 181
Byte:
 (0x0A), Frequency: 181
Byte: d (0x64), Frequency: 169
Byte: - (0x2D), Frequency: 152
Byte: w (0x77), Frequency: 139
Byte: l (0x6C), Frequency: 139
Byte: p (0x70), Frequency: 127
Byte: y (0x79), Frequency: 117
Byte: k (0x6B), Frequency: 89
Byte: b (0x62), Frequency: 89
Byte: j (0x6A), Frequency: 74
Byte: g (0x67), Frequency: 73
Byte: ł (0xB3), Frequency: 59
Byte: , (0x2C), Frequency: 58
Byte: h (0x68), Frequency: 51
Byte: v (0x76), Frequency: 47
Byte: ą (0xB9), Frequency: 42
Byte: ś (0xEA), Frequency: 42
Byte: ť (0x9C), Frequency: 34
Byte: Š (0xE6), Frequency: 27
Byte: ┐ (0xBF), Frequency: 26
Byte: f (0x66), Frequency: 21
Byte: ˇ (0xF3), Frequency: 17
Byte: x (0x78), Frequency: 16
Byte: q (0x71), Frequency: 14
Byte: ˜ (0xF1), Frequency: 7
Byte: B (0x42), Frequency: 5
Byte: C (0x43), Frequency: 5
Byte: S (0x53), Frequency: 3
Byte: ! (0x21), Frequency: 3
Byte: D (0x44), Frequency: 3
Byte: G (0x47), Frequency: 3
Byte: M (0x4D), Frequency: 3
Byte: A (0x41), Frequency: 2
Byte: U (0x55), Frequency: 2
Byte: ? (0x3F), Frequency: 2
Byte: H (0x48), Frequency: 2
Byte: V (0x56), Frequency: 2
Byte: č (0x9F), Frequency: 2
Byte: : (0x3A), Frequency: 1
Byte: . (0x2E), Frequency: 1
Byte: E (0x45), Frequency: 1
Byte: F (0x46), Frequency: 1
Byte: I (0x49), Frequency: 1
Byte: L (0x4C), Frequency: 1
Byte: N (0x4E), Frequency: 1
Byte: O (0x4F), Frequency: 1
```

```
char data: i (0x69)
int code[100]: 1111
int code_length: 4

char data: j (0x6A)
int code[100]: 1110111
int code_length: 7

char data: g (0x67)
int code[100]: 1110110
int code_length: 7

char data: w (0x77)
int code[100]: 111010
int code_length: 6

char data: r (0x72)
int code[100]: 11100
int code_length: 5

char data: t (0x74)
int code[100]: 11011
int code_length: 5

char data: l (0x6C)
int code[100]: 110101
int code_length: 6

char data: p (0x70)
int code[100]: 110100
int code_length: 6

char data: e (0x65)
int code[100]: 1100
int code_length: 4

char data:   (0x20)
int code[100]: 101
int code_length: 3

char data: ť (0x9C)
int code[100]: 10011111
int code_length: 8

char data: ˇ (0xF3)
int code[100]: 100111101
int code_length: 9

char data: x (0x78)
int code[100]: 100111100
int code_length: 9

char data: | (0xB3)
int code[100]: 1001110
int code_length: 7

char data: y (0x79)
int code[100]: 100110
int code_length: 6
```



Do wykonania zadania wykorzystano strukturę modelu informacji, funkcję sortującą oraz kopiec minimalny oraz drzewo z poprzedniego zadania. Na podstawie stworzonej listy słów kodowych tworzony jest ciąg string składający się z samych zer i jedynek. Część tego ciągu to część informacyjna, gdzie dwa pierwsze bajty zawierają informacje o ilości bitów, z których składa się zakodowana wiadomość. Następny bajt zawiera informacje o ilości słów kodowych, następnie znajdują się bajty z informacjami o znakach, długościach słów kodowych tych znaków i występująca po tym bajcie odpowiednia ilość bitów reprezentująca dane słowo kodowe danego znaku. Tak utworzony ciąg dzielę po 8 bitów, przypisuję ich binarnym wartościom odpowiadające tym wartościom znaki i zapisuje je do pliku.

Po skompilowaniu programu oraz podaniu argumentu Teskt.txt, program poprawnie zmienia rozszerzenie i tworzy skompresowany plik o danym rozszerzeniu. W celu potwierdzenia na konsoli wypisywany jest model informacyjny oraz lista słów kodowych. W oknie wypisywana jest zawartość skompresowanego pliku.

Listing programu:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>


////////////

// Stworzenie listy przechowujacej unikatowe znaki i ich czestosci, znajdujace sie w ciagu w przekazanym pliku, stanowi ona nieposortowany model informacji

typedef struct Node {

   char data;

   size_t frequency;

   struct Node* next;

} Node;


Node* createNode(char data) {

   Node* newNode = (Node*)malloc(sizeof(Node));

   newNode->data = data;

   newNode->frequency = 1;

   newNode->next = NULL;
```

```
    return newNode;
}

void insertNode(Node** head, char data) { // dodawanie elementow do modelu w posortowany sposob
    if (*head == NULL) {
        *head = createNode(data);
        return;
    }

    Node* current = *head;
    while (current->next != NULL) {
        if (current->data == data) { // sprawdzanie czy nie powtarza sie dany znak
            current->frequency++;
            return;
        }
        current = current->next;
    }

    if (current->data == data) { // sprawdzenie czy nie powtarza sie znak w ostatnim(lub pierwszym dodanym) node
        current->frequency++;
    }
    else {
        current->next = createNode(data);// tworzenie nowego node'a dla nowego znaku
    }
}

void sortModel(Node** head) { // bubble sort
    if (*head == NULL || (*head)->next == NULL) { // pojedynczy node jest posortowany
        return;
    }

    int swapped;
    Node* current;
```

```
    Node* last = NULL;

    do {
        swapped = 0;
        current = *head;

        while (current->next != last) { // najmniejsza wartosc bedzie wypychana na koniec listy i oznaczona jako last, aby nie trzeba bylo iterowac ponownie po dlugosci calej listy
            if (current->frequency < current->next->frequency) {
                char tempData = current->data;
                size_t tempFrequency = current->frequency;
                current->data = current->next->data;
                current->frequency = current->next->frequency;
                current->next->data = tempData;
                current->next->frequency = tempFrequency;
                swapped = 1;
            }
            current = current->next;
        }
        last = current;
    } while (swapped); // do momentu az dochodzi do zamian, petla sie wykonuje
}

size_t getSize(Node* head) { // policzenie elementow listy (unikatowych znakow)

    Node* current = head;
    size_t iter = 0;
    while (current != NULL) {
        current = current->next;
        iter++;
    }
    return iter;
}
```

```
void freeList(Node** head) { // zwolnienie pamieci
   Node* current = *head;
   while (current != NULL) {
      Node* temp = current;
      current = current->next;
      free(temp);
   }
   *head = NULL;
}
```

/// //

// Kopiec minimalny przechowujacy dane z listy;modelu informacyjnego

// elementy kopca nie sa ze soba polaczone, kazdy element stanowi oddzielny korzen, ktory znajduje sie we wspolnej tablicy wskaznikow array struktury MinHeap

// po zapelnieniu tej tablicy, elementy sa sortowane pod wzgledem ich czestosci, tworzac poprawna strukture kopca minimalnego

// z tak utworzonego kopca minimalnego beda wyciagane dwa elementy o najmniejszych czestosciach(wyciagajac je z wierzcholka kopca)

// dla wyjetych elementow stworzony zostanie nowy, wspolny korzen, przechowujacy sume czestosci jego lisci

// tak utworzone poddrzewo jest zwracane do kopca; tablicy wskaznikow

// proces jest powtarzany az kopiec bedzie skladal sie tylko z jednego elementu;tablica wskaznikow bedzie posiadala tylko jeden element bedacy wskaznikiem na cale drzewo Huffmana

```
typedef struct  MinHeapNode { // element kopca do ktorego beda przekazywane dane z listy

   char data;
   size_t frequency;
   struct MinHeapNode* left, * right;

}MinHeapNode;
```

typedef struct MinHeap { // kopiec posiadajacy tablice wskaznikow na poszczegolne elementy kopca, kopiec poczatkowo zostanie zapelniony danymi z listy, a nastepnie posortowany, tak aby zachowac wlasnosc kopca

```
    size_t size;
    size_t capacity;
    struct MinHeapNode** array;

}MinHeap;



MinHeapNode* createMinHeapNode(Node* list1_var) { // tworzenie elementu kopca kopiujac dane z listy

    MinHeapNode* temp = (MinHeapNode*)malloc(sizeof(struct MinHeapNode));

    temp->left = NULL;
    temp->right = NULL;
    temp->data = list1_var->data;
    temp->frequency = list1_var->frequency;

    return temp;

}

MinHeap* createMinHeap(size_t capacity) { // tworzenie pustego kopca o zadanej pojemnosci i poczatkowym rozmiarze wynoszacym 0

    MinHeap* temp = (MinHeap*)malloc(sizeof(MinHeap));

    temp->size = 0;
    temp->capacity = capacity;
    temp->array = (MinHeapNode**)malloc(temp->capacity * sizeof(MinHeapNode*)); // alokowanie pamieci tablicy wskaznikow na poszczegolne elementy MinHeapNode

    return temp;
}
```

```
void swapMinHeapNode(MinHeapNode** x, MinHeapNode** y) { // przekazanie jako parametry adresow wskaznikow, w celu zamiany wartosci pod tymi adresami; funkcja wykorzystywana przy sortowaniu kopca

    MinHeapNode* temp = *x;
    *x = *y;
    *y = temp;

}

// funkcja przywracajaca wlasnosci kopca minimalnego
// jezeli wyliczony indeks lewego potomka jest mniejszy od rozmiaru kopca i lewy potemek jest mniejszy od poczatkowego rodzica, do indeksu rodzica jest przypisywany indeks lewego potomka
// analogicznie z prawym potomkiem
// jezeli doszlo do zmiany, zamieniane sa elementy kopca i rekurencyjnie proces jest powtarzany startujacac od nowo ustawionego indeksu parent
void minHeapify(MinHeap* _minHeap, size_t index) {

    size_t parent = index;
    size_t left_child = 2 * index + 1;
    size_t right_child = 2 * index + 2;

    if (left_child < _minHeap->size && _minHeap->array[left_child]->frequency < _minHeap->array[parent]->frequency)
        parent = left_child;
    if (right_child < _minHeap->size && _minHeap->array[right_child]->frequency < _minHeap->array[parent]->frequency)
        parent = right_child;
    if (parent != index) {

        swapMinHeapNode(&_minHeap->array[index], &_minHeap->array[parent]);
        minHeapify(_minHeap, parent);
    }
}
```

```
// funkcja wyciagajaca z kopca wierzcholek, wskaznik temp zapamietuje adres wierzcholka, wiercholek jest nadpisywany ostatnia wartoscia kopca
// rozmiar kopca jest zmniejszany(elementy nadal istnieja w tablicy, ale nie beda juz brane pod uwage)
// na koncu minHeapify przywraca wlasnosc kopca minimalnego
MinHeapNode* extractMinHeap(MinHeap* _minHeap) {

    MinHeapNode* temp = _minHeap->array[0];

    _minHeap->array[0] = _minHeap->array[_minHeap->size - 1];
    _minHeap->size--;// dotychczasowy ostatni element tablicy bedzie nieosiagalny, bo juz nie bedzie potrzebny
    minHeapify(_minHeap, 0);

    return temp;
}

// utworzenie kopca o pojemnosci rownej rozmiarowi listy, a nastepnie zapelnienie go elementami tej listy i naprawa kopca
MinHeap* SetUpMinHeap(Node* list1) {


    Node* current = list1;
    size_t capacity = getSize(current);
    MinHeap* _minHeap = createMinHeap(capacity);
    size_t last_index;

    for (int i = 0; i < capacity; i++) {

        _minHeap->array[i] = createMinHeapNode(current);
        current = current->next;
    }

    _minHeap->size = capacity; // zapelniono kopiec
    last_index = _minHeap->size - 1;
```

```
    for (int i = (last_index - 1) / 2; i >= 0; i--) { // przejscie po kazdym korzeniu i naprawa kopca

        minHeapify(_minHeap, i);
    }

    return _minHeap;
}

// drzewo Huffmana utworzone z elementow struktury MinHeapNode, ktore znajduja sie w MinHeap
MinHeapNode* HuffmanTree(MinHeap* _minHeap) {

    MinHeapNode* left_child, * right_child, * parent;
    size_t new_index;

    while (_minHeap->size != 1) {

        left_child = extractMinHeap(_minHeap);
        right_child = extractMinHeap(_minHeap);

        // tworzenie poddrzewa
        parent = (MinHeapNode*)malloc(sizeof(MinHeapNode));
        parent->data = '#';
        parent->frequency = left_child->frequency + right_child->frequency;
        parent->left = left_child;
        parent->right = right_child;

        // ponowne wstawienie utworzonego poddrzewa do kopca
        _minHeap->size++;
        new_index = _minHeap->size - 1;
        _minHeap->array[new_index] = parent;

        for (int i = (new_index - 1) / 2; i >= 0; i--) { // przejscie po kazdym korzeniu i naprawa kopca
```

```
            minHeapify(_minHeap, i);
        }
    }

    return extractMinHeap(_minHeap); // zwrocenie drzewa Huffmana
}

void freeTree(MinHeapNode* root) {
    if (root == NULL) {
        return;
    }
    freeTree(root->left);
    freeTree(root->right);

    free(root);

}

void modify_file_name_extension1(char* str) {
    char* dot_ptr = strchr(str, '.');
    if (dot_ptr != NULL) {
        *dot_ptr = '\0';
        strcat_s(str, strlen(str) + 9, ".Huffman");
    }
}

void printModel(Node* head) {
    printf("\nModel informacyjny:\n");
    Node* current = head;
    while (current != NULL) {
        unsigned int variable = static_cast<unsigned int>(*reinterpret_cast<unsigned char*>(&current->data));
        variable &= 0x000000FF;
```

```
        printf("Byte: %c (0x%02X), Frequency: %zu\n", (current->data || isalnum(current->data) ? current->data : ' '),
variable, current->frequency);
        current = current->next;
    }
    printf("\n----------\n");
}

typedef struct HuffmanCode {

    unsigned char data; // znak
    int code[100]; // slowo kodowe
    int code_length; // dlugosc slowa kodowego
    struct HuffmanCode* next;

}HuffmanCode;

// tworzenie tabeli kodowej, wypisujac po kolei kod kazdego unikatowego znaku do pliku wyjsciowego
void HuffmanCodes(MinHeapNode* root, int buffer[], int index, HuffmanCode** head) {

    // lewe galezie oznaczone jako 0, prawe 1
    if (root->left) {
        buffer[index] = 0;
        HuffmanCodes(root->left, buffer, index + 1, head);
    }
    if (root->right) {
        buffer[index] = 1;
        HuffmanCodes(root->right, buffer, index + 1, head);
    }
    // po dotarciu do liscia wypisywanie danych do listy slow kodowych poszczegolnych znakow
    if (!(root->left) && !(root->right)) {

        // tworzenie nowego node'a pod slowo kodowe
        HuffmanCode* new_code = (HuffmanCode*)malloc(sizeof(HuffmanCode));
```

```
        new_code->data = root->data;
        new_code->code_length = index;
        new_code->next = NULL;

        for (int i = 0; i < new_code->code_length; i++) {
            new_code->code[i] = buffer[i];
        }
        // tworzenie listy
        if (*head == NULL) {
            *head = new_code;
        }
        else {
            new_code->next = *head;
            *head = new_code;
        }
    }
}

void printCodess(HuffmanCode* head) { // testowanie wypisywania listy slow kodowych
    if (head == NULL) {
        return;
    }
    while (head) {
        printf("char data: %c (0x%02X)\n", head->data, head->data);
        printf("int code[100]: ");
        for (int i = 0; i < head->code_length; i++) {
            printf("%d", head->code[i]);
        }
        printf("\nint code_length: %d\n\n", head->code_length);
        head = head->next;
    }
}
```

```
void compressFile(char* argv[]) {

   char input_file_name[100], output_file_name1[100];
   FILE* output_file1;
   FILE* input_file;
   char* input_string;
   // otwarcie pliku wejsciowego i pozyskanie string'a z input_file
   strcpy_s(input_file_name, sizeof(input_file_name), argv[1]);
   errno_t err = fopen_s(&input_file, input_file_name, "rb");
   if (err != 0) {
      printf("Nie udalo sie otworzyc pliku %s\n", input_file_name);
      return;
   }

   fseek(input_file, 0, SEEK_END);
   long size = ftell(input_file);
   fseek(input_file, 0, SEEK_SET);

   input_string = (char*)malloc(size);
   fread_s(input_string, size, 1, size, input_file);

   fclose(input_file);
   ////////////////////////////////////////////////////////////////////////////////////////////////////
  // zapełnienie listy kolejnymi znakami ciągu, lista przechowuje znaki i ich czestosci
   Node* head_list1 = NULL; // model informacyjny

   for (int i = 0; i < size; i++) {

      insertNode(&head_list1, input_string[i]);
   }
```

```
MinHeap* _minHeap = SetUpMinHeap(head_list1);// utworzenie i zapelnienie kopca elementami z listy
MinHeapNode* root = HuffmanTree(_minHeap); // stworzenie  drzewa
////////////////////////////////////////////////////////////////////////////////////////////////////

// wypisywanie do pliku wyjsciowego
// zmiana rozszerzen
modify_file_name_extension1(input_file_name);
strcpy_s(output_file_name1, sizeof(output_file_name1), input_file_name);

sortModel(&head_list1); // sortowanie modelu informacyjnego
int code_buffer[256] = { 0 };
int index = 0;

HuffmanCode* head_list2 = NULL; // lista slow kodowych

HuffmanCodes(root, code_buffer, index, &head_list2);

//############# KOMPRESJA ####################
// Przygotowywanie ciagu do kompresji
char* encode_string = NULL;
char* info_string = NULL;
char* pre_compress_string = NULL;
HuffmanCode* code_iterator = head_list2; // iterator po liscie slow kodowych
int flag = 1;
bool check_first_occurance[256] = { false };

char* encode_string1 = (char*)malloc(9 * sizeof(char)); // 8 bitow + null-terminator; string dla ilosci slow kodowych
int codes_count = getSize(head_list1); // rozmiar modelu informacyjnego jest rowny ilosci slow kodowych
for (int i = 7; i >= 0; i--) { // petla przedstawiajaca codes_count binarnie w postaci stringa
    encode_string1[7 - i] = ((codes_count >> i) & 1) + '0';
}
```

```
    encode_string1[8] = '\0';

    for (int i = 0; i < size; i++) { // iteruje po input_stringu

        while (code_iterator) { // iteruje po liscie slow kodowych

            if ((unsigned char)(input_string[i]) == code_iterator->data) {

                // konwersja slowa kodowego na string

                int single_code_length = code_iterator->code_length; // dlugosc konkretnego slowa kodowego

                char* single_code = (char*)malloc((single_code_length + 1) * sizeof(char)); // string dla slowa kodowego w czesci kodujacej ciag wejsciowy
                for (int i = 0; i < single_code_length; i++) {
                    single_code[i] = code_iterator->code[i] + '0'; // '0' w celu przekazywania znakow ascii, 0 + '0' = '0', 1 + '0' = '1'
                }
                single_code[single_code_length] = '\0'; // utworzono string pojedynczego slowa kodowego


                // tworzenie encode stringa zawierajacy zakodowany ciag wejsciowy z pliku
                if (encode_string == NULL) {
                    encode_string = (char*)malloc((single_code_length + 1) * sizeof(char));
                    strcpy_s(encode_string, (single_code_length + 1) * sizeof(char), single_code); // single_code stanowi pierwszy czlon encode_string
                }
                else {
                    size_t encode_string_length = strlen(encode_string);
                    encode_string = (char*)realloc(encode_string, (encode_string_length + single_code_length + 1) * sizeof(char));
                    strcat_s(encode_string, strlen(encode_string) + strlen(single_code) + 1, single_code);

                }
```

// tworzenie podciagow info stringa zawierajacego informacje o ilosci slow kodowych oraz informacje o znakach i odpowiadajacym im slowom kodowym(przed informacja o slowie kodowym znaku jest rowniez podana dlugosc tego slowa kodowego)

if (check_first_occurance[(unsigned char)(input_string[i])] == false) { // jesli jeden z 256 znakow sie juz pojawil, dalsza czesc kodu jest pomijana,poniewaz informacje o danych znakach maja sie pojawic tylko raz

check_first_occurance[(unsigned char)(input_string[i])] = true; // pojawienie sie znaku zostalo uwzglednione

```
char* encode_string2 = (char*)malloc(9 * sizeof(char));; // string dla znaku
for (int i = 7; i >= 0; i--) {
   encode_string2[7 - i] = ((code_iterator->data >> i) & 1) + '0';
}
encode_string2[8] = '\0';

char* encode_string3 = (char*)malloc(9 * sizeof(char));; // string dla dlugosci slowa kodowego
for (int i = 7; i >= 0; i--) {
   encode_string3[7 - i] = ((single_code_length >> i) & 1) + '0';
}
encode_string3[8] = '\0';
```

char* encode_string4 = (char*)malloc(sizeof(char) * (single_code_length + 1)); // string dla slowa kodowego w czesci informacyjnej

strcpy_s(encode_string4, sizeof(char) * (single_code_length + 1), single_code); // single_code stanowi czesc informacji w info_string

```
// laczenie w jeden podciag informacyjny, ktory dolaczam pozniej do info_string
size_t sub_info_string_length;
char* sub_info_string;
```

if (flag == 1) { // dla pierwszego przypadku ciag informacyjny sklada sie na poczatku z informacji o ilosci slow kodowych, w kolejnych iteracjach informacja ta nie musi sie juz powtarzac

sub_info_string_length = strlen(encode_string1) + strlen(encode_string2) + strlen(encode_string3) + strlen(encode_string4);

```
            sub_info_string = (char*)malloc((sub_info_string_length + 1) * sizeof(char));
            strcpy_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string1);
            strcat_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string2);
            strcat_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string3);
            strcat_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string4);
            flag = 0;
        }
        else {

            sub_info_string_length = strlen(encode_string2) + strlen(encode_string3) + strlen(encode_string4);
            sub_info_string = (char*)malloc((sub_info_string_length + 1) * sizeof(char));
            strcpy_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string2);
            strcat_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string3);
            strcat_s(sub_info_string, (sub_info_string_length + 1) * sizeof(char), encode_string4);
        }



        // tworzenie info_stringa
        if (info_string == NULL) {
            info_string = (char*)malloc((sub_info_string_length + 1) * sizeof(char));
            strcpy_s(info_string, (sub_info_string_length + 1) * sizeof(char), sub_info_string);
        }
        else {
            size_t info_string_length = strlen(info_string);
            info_string = (char*)realloc(info_string, (info_string_length + sub_info_string_length + 1) *
sizeof(char));
            strcat_s(info_string, strlen(info_string) + strlen(sub_info_string) + 1, sub_info_string);
        }

        free(encode_string2);
        free(encode_string3);
        free(encode_string4);
```

```
            }
            free(single_code);
            break;

        }
        code_iterator = code_iterator->next;
    }
    code_iterator = head_list2; // po kazdym break powrot wskaznika na poczatek

}
free(encode_string1);

int encoded_length = strlen(encode_string); // na dwoch bajtach
char* string_encoded_length1 = (char*)malloc(9 * sizeof(char)); // lewy bajt
char* string_encoded_length2 = (char*)malloc(9 * sizeof(char)); // prawy bajt

for (int i = 7; i >= 0; i--) { // petla przedstawiajaca dlugosc zakodowanego ciagu binarnie jako string na dwoch bajtach(tak aby moc kompresowac wieksze pliki)
    string_encoded_length1[7 - i] = ((encoded_length >> (i+8)) & 1) + '0';
}
string_encoded_length1[8] = '\0';

for (int i = 7; i >= 0; i--) {
    string_encoded_length2[7 - i] = ((encoded_length >> i) & 1) + '0';
}
string_encoded_length2[8] = '\0';

pre_compress_string = (char*)malloc((strlen(string_encoded_length1) + strlen(string_encoded_length2) + strlen(info_string) + strlen(encode_string) + 1) * sizeof(char));
strcpy_s(pre_compress_string, (strlen(string_encoded_length1) + strlen(string_encoded_length2) + strlen(info_string) + strlen(encode_string) + 1), string_encoded_length1);
strcat_s(pre_compress_string, (strlen(string_encoded_length1) + strlen(string_encoded_length2) + strlen(info_string) + strlen(encode_string) + 1), string_encoded_length2);
strcat_s(pre_compress_string, (strlen(string_encoded_length1) + strlen(string_encoded_length2) + strlen(info_string) + strlen(encode_string) + 1), info_string);
```

```
    strcat_s(pre_compress_string, (strlen(string_encoded_length1) + strlen(string_encoded_length2) +
strlen(info_string) + strlen(encode_string) + 1), encode_string);
    printf("Ciag przed kompresja: \n%s", pre_compress_string);

    // plik wyjsciowy .Huffman
    err = fopen_s(&output_file1, output_file_name1, "wb");
    if (err != 0) {
        printf("Nie udalo sie utworzyc/otworzyc pliku wyjsciowego  %s\n", output_file_name1);
        return;
    }

    // na tak utworzonym stringu dokonuje podzialu po 8 bitow, kazde kolejne 8 znakow(bajtow) stringa
reprezentujacych bity zapisuje jako jeden bajt i wpisuje do pliku
    unsigned char _buffer = 0;
    int _buffer_index = 0;
    int j;
    for (j = 0; j < strlen(pre_compress_string); j++) {

        unsigned char bit = pre_compress_string[j];
        bit = bit - 48; // '0' = 48 , '1' = 49
        _buffer = _buffer << 1;
        _buffer |= bit; // wstawianie kolejnych cyfr do bajta
        _buffer_index++;

        if (_buffer_index == 8) { // gdy bajt sie zapelni, wysylany jest do pliku
            fwrite(&_buffer, sizeof(_buffer), 1, output_file1);
            printf("\nBuffer nr.%d: (0x%02X)", (j+1)/8, _buffer);
            _buffer = 0;
            _buffer_index = 0;
        }
    }

    // przypadek gdy bajt sie nie zapelni
    if (_buffer_index > 0) {
```

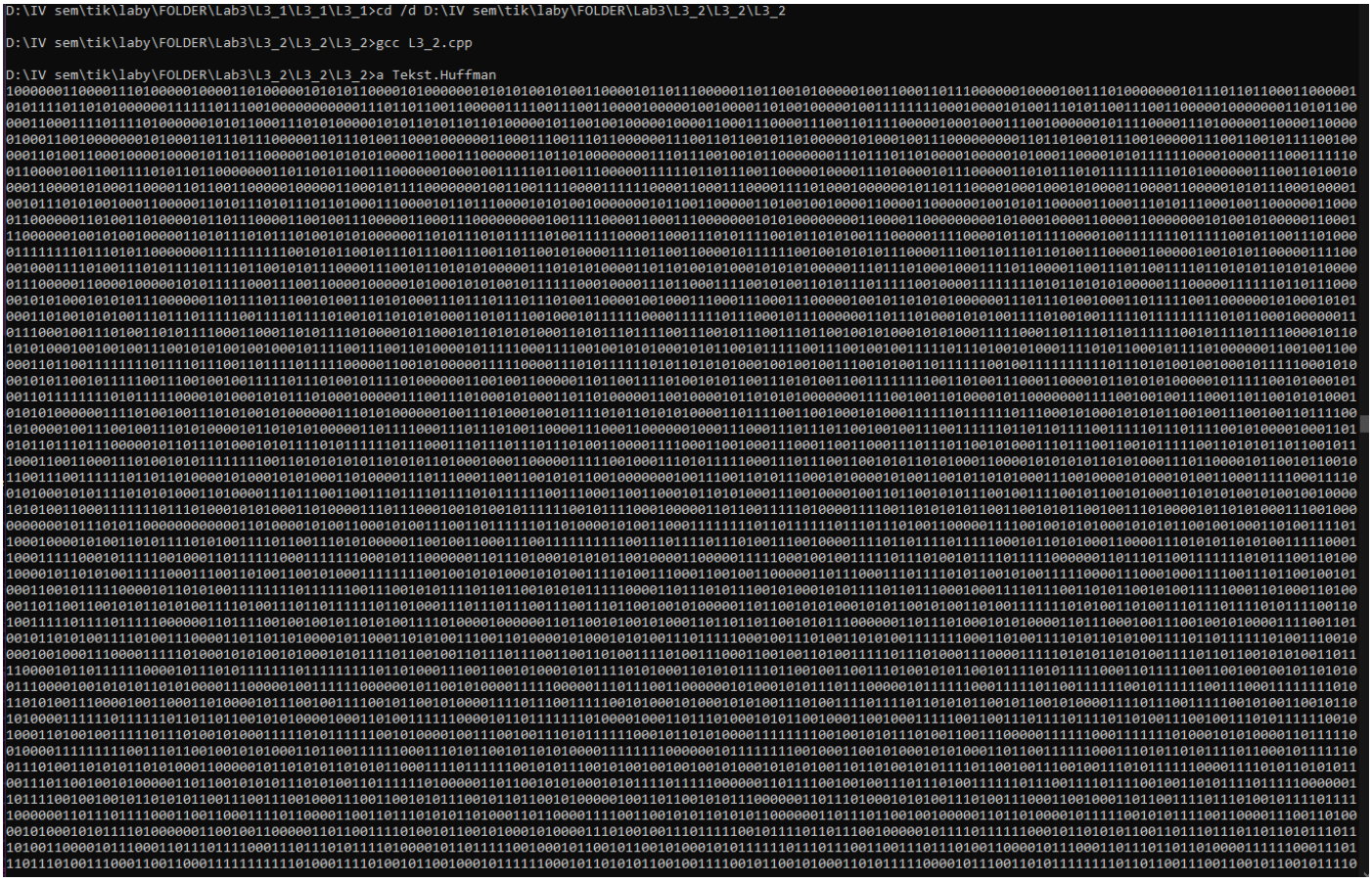
```
        _buffer = _buffer << (8 - _buffer_index); // przesuwanie o tyle miejsc, ile zostalo niezapelnionych
        fwrite(&_buffer, sizeof(_buffer), 1, output_file1);
        printf("\nBuffer dopelniony: (0x%02X)\n", _buffer);
        printf("\nRozmiar przed kompresja: %d\n", size);
        printf("Rozmiar po kompresji: %d\n", ((j + 1) / 8) + 1);
    }
    else {
        printf("\nRozmiar przed kompresja: %d\n", size);
        printf("Rozmiar po kompresji: %d\n", ((j + 1) / 8));
    }


    fclose(output_file1);


    printModel(head_list1);
    printCodess(head_list2);

    freeList(&head_list1);
    freeTree(root);
    root = NULL;
    free(_minHeap);
    _minHeap = NULL;
    free(encode_string);
    free(info_string);
    free(pre_compress_string);

}

int main(int argc, char* argv[]) {
```

```
    if (argc != 2) {
        printf("Poprawna forma: %s <nazwa_pliku>\n", argv[0]);
        return 1;
    }

    compressFile(argv);




    return 0;
}
```

Zadanie L2_2

```
/
ĆWICZENIE LABORATORYNE NUMER 3. ZADANIE 2
DEKOMPRESJA PLIKU

WYNIK PROGRAMU: PLIK ODTWORZONY
ARGUMENT1: NAZWA PLIKU WEJŚCIOWEGO (nazwa kropka rozszerzenie)
PLIK WYJŚCIOWY  MA TĘ SAMĄ nazwę PLIKU WEJŚCIOWEGO ZE ZMIENIONYM PRZEZ PROGRAM
rozszerzeniem (*.recovery)
ZAPIS DO PLIKU: BINARNY
Weryfikacja poprawności kodowania po dekompresji przez porównanie z oryginałem
*/
```

```
D:\IV sem\tik\laby\FOLDER\Lab3\L3_1\L3_1>cd /d D:\IV sem\tik\laby\FOLDER\Lab3\L3_2\L3_2

D:\IV sem\tik\laby\FOLDER\Lab3\L3_2\L3_2>gcc L3_2.cpp

D:\IV sem\tik\laby\FOLDER\Lab3\L3_2\L3_2>a Tekst.Huffman
```



```
Liczba bitow final stringa: 34768
message size: 33159
info count: 65
ZDEKODOWANY CIAG:
Output length: 7070
Sentencje |aci˜skie z t|umaczeniem Autor: zdenkowicz A
ab ovo - od pocz╣tku
absens carens - nieobecny traci
absit! - Uchowaj, Bo┐e!
ad patres - do ojc˜w
alea iacta est - ko┼ci zosta|y rzucone
alter ego - drugie ja
amor omnibus idem - mi|o┼Ü dla wszystkich jednaka.
amor vincit omnia - mi|o┼Ü wszystko zwyci┐a
anima vilis - pod|a dusza
asinus asinorum - osio| nad os|ami
aurea dicta - z|ote s|owa
ave, Caesar, morituri te salutant - witaj, Cezarze, maj╣cy umrze┼Ü ci┼Ľ
pozdrawiaj╣
B
bene meritus - dobrze zas|u┐ony
bona fide - w dobrej wierze
bonum ex malo non fit - dobro nie rodzi si┼Ľ ze z|a
boss lassus fortius figit pedem - zm┼Ľczony w˜| szybciej pracuje
C
caeca invidia est - zazdro┼Ü jest ┼Ľlepa
carpe diem - korzystaj z ka┐dego dnia
cicer cum caule - groch z kapust╣
cogito, ergo sum - my┼Ľl┼Ľ, wi┼Ľc jestem
consummatum est - sta|o si┼Ľ
corvus albus - bia|y kruk
cui bono? - na czyj╣ korzy┼Ü?
cum ventis litigare - walczy┼Ü z wiatrem
D
dances macabres - taniec ┼Ľmierci
Deus ex machina - b˜g z machiny
dictum sapienti sat - m╣drej g|owie dot┼Ü dwie s|owie
diem perdidi - straci|em dzie˜
dies irae - dzie˜ gniewu
E
ecce homo - oto cz|owiek
ego sum qui sum - jestem, kt˜ry jestem
errare humanum est - b|╣dzi┼Ü jest rzecz╣ ludzk╣
est modus in rebus - wszystko ma swoje granice
et tu, Brute, contra me - i ty, Brutusie, przeciwko mnie
ex malis eligere minima - wybiera┼Ü mniejsze z|o
ex oriente lux - ┼Ľwiat|o ze wschodu
experto credite - wierzcie do┼Ľwiadczonemu
F
faber est quisque suae fortunae - ka┐dy jest kowalem w|asnego losu
felix culpa - szczę┼Ľliwa wina
ferro ignique - ogniem i mieczem
festina lente - spiesz si┼Ľ powoli
G
gallus in suo sterquilinio plurimum potest - kogut tylko na w|asnym gnojowisku
du┐o mo┐e
genius loci - duch opieku˜czy miejsca
gladiator in arena consilium capit - gladiator decyduje dopiero na arenie
gloria victis - chwa|a zwyci┼Ľ┐onym
gloria virtuti resonat - s|awa jest echem cnoty
graviora manent - najgorsze dopiero nadejdzie
H
historia magistra vitae - historia nauczycielk╣ ┐ycia
hodie mihi, cras tibi - dzi┼Ľ mnie, jutro tobie
hominis est errare, insipientis in errore perseverare - ludzk╣ rzecz╣ jest
b|╣dzi┼Ü, g|upc˜w trwa┼Ü w b|┼Ľdzie
homo homini lupus est - cz|owiek cz|owiekowi jest wilkiem
homo sacra res homini - cz|owiek jest rzecz╣ ┼Ľwi┼Ľt╣ dla cz|owieka
horribile dictu - strach powiedzie┼Ü
hospes, hostis - ka┐dy obcy to wr˜g
I
ignavis sepmer feriae - lenie zawsze maj╣ ┼Ľwi┼Ľto
ignis non exstinguitur igne - ognia nie gasi si┼Ľ ogniem
impares nascimur, pares morimur - rodzimy si┼Ľ nier˜wni, umieramy r˜wni
imperare sibi maximum est imperium - panowa┼Ü nad sob╣ to najwy┐sza w|adza
impos animi - s|aby na umy┼Ťle
impossibilium nulla obligatio est - nikt nie jest zobowi╣zany do rzeczy
niemo┐liwych
in articulo mortis - w obliczu ┼Ľmierci
in principio erat Verbum - na pocz╣tku by|o S|owo
in vino veritas - wino rozwi╣zuje j┼Ľzyk
L
labor omnia vincit - praca wszystko przezwyci┼Ľ┐a
laudant, quod non intelligunt - chwal╣ to, czego nie rozumiej╣
lavare manus - umywa┼Ü r┼Ľce
M
magnum in parvo - wiele tre┼Ľci w kr˜tkiej wypowiedzi
manus manum lavat - r┼Ľka r┼Ľk┼Ľ myje
```

```
D:\IV sem\tik\laby\FOLDER\Lab3\L3_2\L3_2\L3_2>type Tekst.recovery

Sentencje |aci˝skie z t|umaczeniem Autor: zdenkowicz A
ab ovo - od pocz╣tku
absens carens - nieobecny traci
absit! - Uchowaj, Bo┐e!
ad patres - do ojc˝w
alea iacta est - ko┼ci zosta|y rzucone
alter ego - drugie ja
amor omnibus idem - mi|o┼Š dla wszystkich jednaka.
amor vincit omnia - mi|o┼Š wszystko zwyci┐a
anima vilis - pod|a dusza
asinus asinorum - osio| nad os|ami
aurea dicta - z|ote s|owa
ave, Caesar, morituri te salutant - witaj, Cezarze, maj╣cy umrze┼Š ci┌
pozdrawiaj╣
B
bene meritus - dobrze zas|u┐ony
bona fide - w dobrej wierze
bonum ex malo non fit - dobro nie rodzi si┌ ze z|a
boss lassus fortius figit pedem - zm┌czony w˝| szybciej pracuje
C
caeca invidia est - zazdro┼Š jest ┼Blepa
carpe diem - korzystaj z ka┐dego dnia
cicer cum caule - groch z kapust╣
cogito, ergo sum - my┼Bl┌, wi┌c jestem
consummatum est - sta|o si┌
corvus albus - bia|y kruk
cui bono? - na czyj╣ korzy┼Š?
cum ventis litigare - walczy┼Š z wiatrem
D
dances macabres - taniec ┼Bmierci
Deus ex machina - b˝g z machiny
dictum sapienti sat - m╣drej g|owie dot┼Š dwie s|owie
diem perdidi - straci|em dzie˝
dies irae - dzie˝ gniewu
E
ecce homo - oto cz|owiek
ego sum qui sum - jestem, kt˝ry jestem
errare humanum est - b|╣dzi┼Š jest rzecz╣ ludzk╣
est modus in rebus - wszystko ma swoje granice
et tu, Brute, contra me - i ty, Brutusie, przeciwko mnie
ex malis eligere minima - wybiera┼Š mniejsze z|o
ex oriente lux - ┼Bwiat|o ze wschodu
experto credite - wierzcie do┼Bwiadczonemu
F
faber est quisque suae fortunae - ka┐dy jest kowalem w|asnego losu
felix culpa - szczę┼Bliwa wina
ferro ignique - ogniem i mieczem
festina lente - spiesz si┌ powoli
G
gallus in suo sterquilinio plurimum potest - kogut tylko na w|asnym gnojowisku
du┐o mo┐e
genius loci - duch opieku˝czy miejsca
gladiator in arena consilium capit - gladiator decyduje dopiero na arenie
gloria victis - chwa|a zwyci┐onym
gloria virtuti resonat - s|awa jest echem cnoty
graviora manent - najgorsze dopiero nadejdzie
H
historia magistra vitae - historia nauczycielk╣ ┐ycia
hodie mihi, cras tibi - dzi┼A mnie, jutro tobie
hominis est errare, insipientis in errore perseverare - ludzk╣ rzecz╣ jest
b|╣dzi┼A, g|upc˝w trwa┼Š w b|┌dzie
homo homini lupus est - cz|owiek cz|owiekowi jest wilkiem
homo sacra res homini - cz|owiek jest rzecz╣ ┼Bwi┌t╣ dla cz|owieka
horribile dictu - strach powiedzie┼Š
hospes, hostis - ka┐dy obcy to wr˝g
I
ignavis sepmer feriae - lenie zawsze maj╣ ┼Bwi┌to
ignis non exstinguitur igne - ognia nie gasi si┌ ogniem
impares nascimur, pares morimur - rodzimy si┌ nier˝wni, umieramy r˝wni
imperare sibi maximum est imperium - panowa┼Š nad sob╣ to najwy┐sza w|adza
impos animi - s|aby na umy┼Ble
impossibilium nulla obligatio est - nikt nie jest zobowi╣zany do rzeczy
```

Zakodowany ciag w postaci znakow odpowiadających wartościom bajtow dekoduję spowrotem na ciąg string składający się z zer i jedynek. Analogicznie do zadania pierwszego, przechodząc po ciągu wyciągam kolejno informacje o długości zakodowanego ciągu, ilości słów kodowych, danych znakach, długościach ich słów kodowych i same słowa kodowe, a następnie na tej podstawie odkodowuje rzeczywisty zakodowany ciąg.

Po skompilowaniu programu oraz podaniu argumentu Tekst.Huffman z poprzedniego zadania, program poprawnie tworzy plik o rozszerzeniu .recovery. Sam program wypisuje zawartość odzyskanego pliku. Również za pomocą type wypisywana jest zawartość odzyskanego pliku. Wypisywany jest również na początku odzyskany ciąg zer i jedynek na podstawie, którego będzie dekodowana oryginalna wiadomość.

Listing programu:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>


void modify_file_name_extension1(char* str) {
    char* dot_ptr = strchr(str, '.');
    if (dot_ptr != NULL) {
        *dot_ptr = '\0';
        strcat_s(str, strlen(str) + 10, ".recovery");
    }
}

typedef struct HuffmanCode {

    unsigned char data; // znak
    int code[100]; // slowo kodowe
    int code_length; // dlugosc slowa kodowego
    struct HuffmanCode* next;

}HuffmanCode;

HuffmanCode* createNode(char data, int code_length,int arr[]) {
    HuffmanCode* newNode = (HuffmanCode*)malloc(sizeof(HuffmanCode));
    newNode->data = data;
    newNode->code_length = code_length;

    for (int i = 0; i < code_length; i++) {
        newNode->code[i] = arr[i];
    }


    newNode->next = NULL;
    return newNode;
}

void insertNode(HuffmanCode** head, char data, int code_length, int arr[]) { // dodawanie
elementow do modelu w posortowany sposob
    if (*head == NULL) {
        *head = createNode(data, code_length, arr);
        return;
    }

    HuffmanCode* current = *head;
    while (current->next != NULL) {
        if (current->data == data) { // sprawdzanie czy nie powtarza sie dany znak
```

```
            return;
        }
        current = current->next;
    }

    if (current->data == data) { // sprawdzenie czy nie powtarza sie znak w ostatnim(lub
pierwszym dodanym) node
        return;
    }
    else {
        current->next = createNode(data, code_length, arr);// tworzenie nowego node'a dla
nowego znaku
    }
}


void printCodess(HuffmanCode* head) { // testowanie wypisywania listy slow kodowych
    if (head == NULL) {
        return;
    }
    while (head) {
        printf("char data: %c (0x%02X)\n", head->data, head->data);
        printf("int code[100]: ");
        for (int i = 0; i < head->code_length; i++) {
            printf("%d", head->code[i]);
        }
        printf("\nint code_length: %d\n\n", head->code_length);
        head = head->next;
    }
}

int findMaxCodeLength(HuffmanCode* head) { // zwracanie maksymalnej dlugosci slowa kodowego
    int maxCodeLength = 0;

    while (head != NULL) {
        if (head->code_length > maxCodeLength) {
            maxCodeLength = head->code_length;
        }
        head = head->next;
    }

    return maxCodeLength;
}


void decompressFile(char* argv[]) {

    char input_file_name[100], output_file_name1[100];
    FILE* output_file1;
    FILE* input_file;
    char* input_string;
    // otwarcie pliku wejsciowego i pozyskanie string'a z input_file
    strcpy_s(input_file_name, sizeof(input_file_name), argv[1]);
    errno_t err = fopen_s(&input_file, input_file_name, "rb");
    if (err != 0) {
        printf("Nie udalo sie otworzyc pliku %s\n", input_file_name);
        return;
    }

    fseek(input_file, 0, SEEK_END);
    long size = ftell(input_file); // rozmiar pliku rowny rozmiarowi zakodowanej wiadomosci
    fseek(input_file, 0, SEEK_SET);

    input_string = (char*)malloc(size);
    fread_s(input_string, size, 1, size, input_file);

    fclose(input_file);
```

```
    // wypisywanie do pliku wyjsciowego
    // zmiana rozszerzen
    modify_file_name_extension1(input_file_name);
    strcpy_s(output_file_name1, sizeof(output_file_name1), input_file_name);

    HuffmanCode* head_list2 = NULL; // lista slow kodowych

    //############ DEKOMPRESJA ####################
    // Przeksztalcenie bajtow na bity i zapisanie ciagu bitow jako string

    char* buffer = (char*)malloc(9 * sizeof(char)); // miejsce na przeksztalcenie odczytanego
bajtu na 8 bitow jako znaki w stringu
    char* final_string = NULL;
    for (int i = 0; i < size; i++) { // iteruje po calej zakodowanej wiadomosci

        for (int j = 7; j >= 0; j--) { // petla przedstawiajaca dlugosc zakodowanego ciagu
binarnie jako string
            buffer[7 - j] = ((input_string[i] >> j) & 1) + '0';
        }
        buffer[8] = '\0';

        if (final_string == NULL) {
            final_string = (char*)malloc((8 * size * sizeof(char)) + 1);
            strcpy_s(final_string, (8 * size * sizeof(char)) + 1, buffer);
        }
        else {
            strcat_s(final_string, (8 * size * sizeof(char)) + 1, buffer);
        }

    }
    printf("%s\n", final_string);
    printf("Liczba bitow final stringa: %ld\n", strlen(final_string));

    // plik wyjsciowy .recovery
    err = fopen_s(&output_file1, output_file_name1, "wb");
    if (err != 0) {
        printf("Nie udalo sie utworzyc/otworzyc pliku wyjsciowego  %s\n", output_file_name1);
        return;
    }

    int message_size = 0; // dlugosc zakodowanej wiadomosci w bitach
    int info_count = 0; // liczba slow kodowych
    int current_bit = 0;

    // odczytanie czesci informacyjnej
    for (int i = 0; i < 16; i++) {
        message_size = (message_size << 1) + (final_string[i] - '0');
    }
    printf("message size: %d\n", message_size);
    current_bit += 16;

    for (int i = 0; i < 8; i++) {
        info_count = (info_count << 1) + (final_string[current_bit + i] - '0');
    }
    printf("info count: %d\n", info_count);
    current_bit += 8;


    HuffmanCode* head = NULL;
    int code[100];

    for (int i = 0; i < info_count; i++) {

        unsigned char data = 0;
        int code_length = 0;
        for (int j = 0; j < 8; j++) {
```

```
            data = (data << 1) + (final_string[current_bit + j] - '0');
        }
        current_bit += 8;
        for (int j = 0; j < 8; j++) {
            code_length = (code_length << 1) + (final_string[current_bit + j] - '0');
        }
        current_bit += 8;
        for (int j = 0; j < code_length; j++) {
            code[j] = (final_string[current_bit + j] - '0');
        }
        current_bit += code_length;

        insertNode(&head, data, code_length, code);

    }
    // odczytanie zakodowanej wiadomosci

    HuffmanCode* current = head;

    int max_code_length = findMaxCodeLength(head);
    unsigned char* output_string = NULL;
    long output_length = 0;

    char* temp_string = (char*)malloc(sizeof(char) * max_code_length + 1);
    temp_string[max_code_length] = '\0';
    int found = 0;
    int y = -1;

    for (int i = 0; i < message_size; i++) {


       if (found == 1) {
           temp_string = (char*)malloc(sizeof(char) * max_code_length + 1);
           y = 0;
           found = 0;
       }
       else {
           y++;
       }
       temp_string[y] = final_string[current_bit];
       temp_string[y + 1] = '\0';

       while (current) {
            char* code_string = (char*)malloc((sizeof(char) * (current->code_length + 1))); //
co iteracje zamieniany jest int code na string i porownywany z temp
            for (int k = 0; k < current->code_length; k++) {
                code_string[k] = current->code[k] + '0';
            }
            code_string[current->code_length] = '\0';


            if (strncmp(temp_string, code_string, current->code_length) == 0) { // porownywane
sa stringi do dlugosci stringa z listy slow kodowych, jesli oba stringi sa takie same, zwalnia
sie pamiec temp i przechodzi sie do nastepnego bitu

                output_string = (unsigned char*)realloc(output_string, (output_length + 2) *
sizeof(unsigned char));
                output_string[output_length] = current->data;
                output_string[output_length + 1] = '\0';

                output_length++;
                found = 1;
                free(temp_string);
                temp_string = NULL;
                break;
            }
            free(code_string);
```

```
            current = current->next;
        }
        current_bit++;
        current = head;
        if (i == message_size)
            free(temp_string);

    }
    printf("ZDEKODOWANY CIAG: \n");
    printf("Output length: %ld", output_length);
    int x = 0;
    while (x < output_length) {
        printf("%c", output_string[x]);

        x++;
    }
    printf("\n\n");
    fwrite(output_string, sizeof(unsigned char), output_length, output_file1); // wypisuje po
bajcie az do output_length elementow

    fclose(output_file1);

    //printCodess(head);
    printf("\nZdekompresowano plik");
    free(buffer);
    free(output_string);

}

int main(int argc, char* argv[]) {

    if (argc != 2) {
        printf("Poprawna forma: %s <nazwa_pliku>\n", argv[0]);
        return 1;
    }

    decompressFile(argv);

    return 0;
}
```