Kompresja Huffman'a

1.0

Wygenerowano przez Doxygen 1.9.6

Rozdział 1

Indeks hierarchiczny

1.1 Hierarchia klas

Ta lista dziedziczenia posortowana jest z grubsza, choć nie całkowicie, alfabetycznie:

Rozdział 2

Indeks klas

2.1 Lista klas

Tutaj znajdują się klasy, struktury, unie i interfejsy wraz z ich krótkimi opisami:

Rozdział 3

Indeks plików

3.1 Lista plików

Tutaj znajduje się lista wszystkich udokumentowanych plików z ich krótkimi opisami:

Rozdział 4

Dokumentacja klas

4.1 Dokumentacja klasy bit_file_io

Klasa opakowująca std::fstream. Umożliwia pisanie i czytanie, z i do pliku, pojedyczych bitów.

`#include <bit_file_io.h>`

Metody publiczne

- bit_file_io (std::fstream &file_stream, size_t read_buff_size, size_t write_buff_size)
- void write_bit (uint8_t bit)

  Pisze pojedynczy bit do pliku.
- void flush_bit_buffer ()

  Powoduje wypisanie i wyczyszczenie buforu bitów.
- void flush_buffer ()

  Powoduje wypisanie i wyczyszczenie buforu.
- bool read_bit (uint8_t &bit)

  Przeczytaj pojedynczy bit z pliku.
- bit_file_io & operator<< (const uint8_t bit)
- bool operator>> (uint8_t &bit)

4.1.1 Opis szczegółowy

Klasa opakowująca std::fstream. Umożliwia pisanie i czytanie, z i do pliku, pojedyczych bitów.

Definicja w linii 10 pliku bit_file_io.h.

4.1.2 Dokumentacja konstruktora i destruktora

4.1.2.1 bit_file_io()

```
bit_file_io::bit_file_io (
            std::fstream & file_stream,
            size_t read_buff_size,
            size_t write_buff_size )
```

Definicja w linii 5 pliku bit_file_io.cpp.

4.1.2.2 ~bit_file_io()

```
bit_file_io::~bit_file_io ( )
```

Definicja w linii 15 pliku bit_file_io.cpp.

4.1.3 Dokumentacja funkcji składowych

4.1.3.1 flush_bit_buffer()

```
void bit_file_io::flush_bit_buffer ( )
```

Powoduje wypisanie i wyczyszczenie buforu bitów.

Causes w_bit_buf_ to be saved to w_buff_. If w_buff_ achieves max size it's being flushed.

Definicja w linii 37 pliku bit_file_io.cpp.

4.1.3.2 flush_buffer()

```
void bit_file_io::flush_buffer ( )
```

Powoduje wypisanie i wyczyszczenie buforu.

Causes w_buff_ to be written to file. Any data in w_bit_buf_ is ignored.

Definicja w linii 54 pliku bit_file_io.cpp.

4.1.3.3 operator<<()

```
bit_file_io & bit_file_io::operator<< (
            const uint8_t bit )
```

Definicja w linii 86 pliku bit_file_io.cpp.

4.1.3.4 operator>>()

```
bool bit_file_io::operator>> (
            uint8_t & bit )
```

Definicja w linii 92 pliku bit_file_io.cpp.

4.1.3.5 read_bit()

```
bool bit_file_io::read_bit (
            uint8_t & bit )
```

Przeczytaj pojedynczy bit z pliku.

Parametry

out	*bit*	- bit który zostanie przeczytany

Zwraca

true - kiedy bit został przeczytany

false - kiedy nie ma już bitów do przeczytania

Definicja w linii 64 pliku bit_file_io.cpp.

4.1.3.6 write_bit()

```
void bit_file_io::write_bit (
            uint8_t bit )
```

Pisze pojedynczy bit do pliku.

Parametry

bit	- bit który zostanie wypisany do pliku

Definicja w linii 21 pliku bit_file_io.cpp.

Dokumentacja dla tej klasy została wygenerowana z plików:

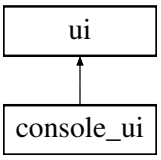
- inc/bit_file_io.h
- src/bit_file_io.cpp

4.2 Dokumentacja klasy console_ui

Prosta implementacja konsolowego interfejsu użytkownika. Pisze na standardowe wyjście i wyjście błędu.

`#include <ui.h>`

Diagram dziedziczenia dla console_ui



Metody publiczne

- void write_message (const std::string &msg) const override

  Wyświetla komunikat na stdout.
- void app_error (const std::string &error_msg) const override

  Wypisuje błąd na stderr i kończy działanie programu z kodem EXIT_FAILURE.

- virtual void write_message (const std::string &msg) const =0

  Wyświetla komunikat użytkownikowi.
- virtual void app_error (const std::string &error_msg) const =0

  Obsługuje błąd aplikacji.

4.2.1 Opis szczegółowy

Prosta implementacja konsolowego interfejsu użytkownika. Pisze na standardowe wyjście i wyjście błędu.

Definicja w linii 28 pliku ui.h.

4.2.2 Dokumentacja funkcji składowych

4.2.2.1 app_error()

```
void console_ui::app_error (
            const std::string & error_msg ) const  [override], [virtual]
```

Wypisuje błąd na stderr i kończy działanie programu z kodem EXIT_FAILURE.

Parametry

error_msg	- komunikat błędu

Implementuje ui.

Definicja w linii 10 pliku ui.cpp.

4.2.2.2 write_message()

```
void console_ui::write_message (
            const std::string & msg ) const  [override], [virtual]
```

Wyświetla komunikat na stdout.

Parametry

msg	- komunikat do wyświetlenia

Implementuje ui.

Definicja w linii 5 pliku ui.cpp.

Dokumentacja dla tej klasy została wygenerowana z plików:

- inc/ui.h
- src/ui.cpp

4.3 Dokumentacja klasy freq_map

Klasa pomocnicza do przechowywania częstotliwości wystepowania bajtów.

```
#include <huffman_tree.h>
```

Metody publiczne

- uint64_t get (uint8_t byte) const

  Pobiera ilość występowania danego bajtu.
- void set (uint8_t byte, uint64_t value)

  Ustawia ilość wystepowania danego bajtu na podaną wartość
- void inc (uint8_t byte)

  Inkrementuje częstotliwość danego bajtu.
- uint16_t size () const

  Zwraca ilość unikatowych bajtów.

4.3.1 Opis szczegółowy

Klasa pomocnicza do przechowywania częstotliwości wystepowania bajtów.

Definicja w linii 9 pliku huffman_tree.h.

4.3.2 Dokumentacja konstruktora i destruktora

4.3.2.1 freq_map()

```
freq_map::freq_map ( )  [inline]
```

Definicja w linii 15 pliku huffman_tree.h.

4.3.3 Dokumentacja funkcji składowych

4.3.3.1 get()

```
uint64_t freq_map::get (
            uint8_t byte ) const  [inline]
```

Pobiera ilość występowania danego bajtu.

Parametry

byte	bajt

Zwraca

uint64_t częstotliwość bajtu byte

Definicja w linii 23 pliku huffman_tree.h.

4.3.3.2 inc()

```
void freq_map::inc (
            uint8_t byte )  [inline]
```

Inkrementuje częstotliwość danego bajtu.

Parametry

byte	byte

Definicja w linii 38 pliku huffman_tree.h.

4.3.3.3 set()

```
void freq_map::set (
            uint8_t byte,
            uint64_t value )  [inline]
```

Ustawia ilość wystepowania danego bajtu na podaną wartość

Parametry

byte	bajt
value	nowa ilość

Definicja w linii 31 pliku huffman_tree.h.

4.3.3.4 size()

```
uint16_t freq_map::size ( ) const  [inline]
```

Zwraca ilość unikatowych bajtów.

Zwraca

uint16_t - ilość unikatowych bajtów

Definicja w linii 44 pliku huffman_tree.h.

Dokumentacja dla tej klasy została wygenerowana z pliku:

- inc/huffman_tree.h

4.4 Dokumentacja klasy huffman_encoder

Klasa służąca do kompresji/dekompresji plików przy pomocy kodowania Huffmana.

```
#include <huffman_encoder.h>
```

Metody publiczne

- huffman_encoder (std::string input_file, std::string output_file, const ui &ui, const size_t buffer_size=size_↩ 16_mb)

  Tworzy nowy obiekt encodera.
- void compress_file ()

  Funkcja kompresująca plik.
- void decompress_file ()

  Funkcja dekompresująca plik.

4.4.1 Opis szczegółowy

Klasa służąca do kompresji/dekompresji plików przy pomocy kodowania Huffmana.

Definicja w linii 12 pliku huffman_encoder.h.

4.4.2 Dokumentacja konstruktora i destruktora

4.4.2.1 huffman_encoder()

```
huffman_encoder::huffman_encoder (
            std::string input_file,
            std::string output_file,
            const ui & ui,
            const size_t buffer_size = size_16_mb )
```

Tworzy nowy obiekt encodera.

Parametry

input_file	- ścierzka do pliku wejściowego
output_file	- ścierzka do pliku wyjściowego
ui	- implementacja interfejsu użytkownika
buffer_size	- rozmiar wewnętrznego bufora

Definicja w linii 16 pliku huffman_encoder.cpp.

4.4.2.2 ~huffman_encoder()

```
huffman_encoder::~huffman_encoder ( )
```

Definicja w linii 31 pliku huffman_encoder.cpp.

4.4.3 Dokumentacja funkcji składowych

4.4.3.1 compress_file()

```
void huffman_encoder::compress_file ( )
```

Funkcja kompresująca plik.

Definicja w linii 33 pliku huffman_encoder.cpp.

4.4.3.2 decompress_file()

```
void huffman_encoder::decompress_file ( )
```

Funkcja dekompresująca plik.

Definicja w linii 127 pliku huffman_encoder.cpp.

Dokumentacja dla tej klasy została wygenerowana z plików:

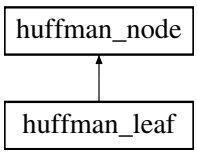
- inc/huffman_encoder.h
- src/huffman_encoder.cpp

4.5 Dokumentacja klasy huffman_leaf

Liść drzewa Huffmana. Zawiera częstotliwość oraz bajt, który reprezentuje. Dziedziczy po huffman_node.

```
#include <huffman_tree.h>
```

Diagram dziedziczenia dla huffman_leaf



Metody publiczne

- huffman_leaf (const uint64_t frequency, const uint8_t value)
- uint8_t get_value () const

  Zwraca bajt reprezentowany przez liść

Metody publiczne dziedziczone z huffman_node

- huffman_node (const uint64_t frequency, huffman_node *left_child, huffman_node *right_child)
- uint64_t get_frequency () const

  Pobiera częstotlowść przechowywaną przez wierzchołek.
- huffman_node * get_left_child () const

  Zwraca lewe dziecko.
- huffman_node * get_right_child () const

  Zwraca prawe dziecko.

4.5.1 Opis szczegółowy

Liść drzewa Huffmana. Zawiera częstotliwość oraz bajt, który reprezentuje. Dziedziczy po huffman_node.

Definicja w linii 94 pliku huffman_tree.h.

4.5.2 Dokumentacja konstruktora i destruktora

4.5.2.1 huffman_leaf()

```
huffman_leaf::huffman_leaf (
            const uint64_t frequency,
            const uint8_t value )  [inline]
```

Definicja w linii 100 pliku huffman_tree.h.

4.5.3 Dokumentacja funkcji składowych

4.5.3.1 get_value()

```
uint8_t huffman_leaf::get_value ( ) const  [inline]
```

Zwraca bajt reprezentowany przez liść

Zwraca

uint8_t - bajt reprezentowany przez liść

Definicja w linii 109 pliku huffman_tree.h.

Dokumentacja dla tej klasy została wygenerowana z pliku:

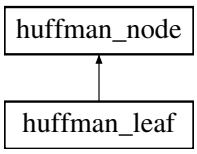
- inc/huffman_tree.h

4.6 Dokumentacja klasy huffman_node

Wierzchołek drzewa Huffmana.

`#include <huffman_tree.h>`

Diagram dziedziczenia dla huffman_node



Metody publiczne

- huffman_node (const uint64_t frequency, huffman_node ∗left_child, huffman_node ∗right_child)
- uint64_t get_frequency () const

  Pobiera częstotlowść przechowywaną przez wierzchołek.
- huffman_node ∗ get_left_child () const

  Zwraca lewe dziecko.
- huffman_node ∗ get_right_child () const

  Zwraca prawe dziecko.

4.6.1 Opis szczegółowy

Wierzchołek drzewa Huffmana.

Definicja w linii 57 pliku huffman_tree.h.

4.6.2 Dokumentacja konstruktora i destruktora

4.6.2.1 huffman_node()

```
huffman_node::huffman_node (
            const uint64_t frequency,
            huffman_node * left_child,
            huffman_node * right_child )  [inline]
```

Definicja w linii 64 pliku huffman_tree.h.

4.6.3 Dokumentacja funkcji składowych

4.6.3.1 get_frequency()

```
uint64_t huffman_node::get_frequency ( ) const  [inline]
```

Pobiera częstotlowść przechowywaną przez wierzchołek.

Zwraca

uint64_t - częstotliwość

Definicja w linii 75 pliku huffman_tree.h.

4.6.3.2 get_left_child()

```
huffman_node * huffman_node::get_left_child ( ) const  [inline]
```

Zwraca lewe dziecko.

Zwraca

huffman_node* - lewe dziecko

Definicja w linii 81 pliku huffman_tree.h.

4.6.3.3 get_right_child()

```
huffman_node * huffman_node::get_right_child ( ) const  [inline]
```

Zwraca prawe dziecko.

Zwraca

huffman_node* - prawe dziecko

Definicja w linii 87 pliku huffman_tree.h.

Dokumentacja dla tej klasy została wygenerowana z pliku:

- inc/huffman_tree.h

4.7 Dokumentacja klasy huffman_tree

Reprezentacja drzewa Huffmana.

```
#include <huffman_tree.h>
```

Metody publiczne

- huffman_tree (const freq_map &chars_freq)

  Tworzy drzewo Huffmana oraz generuje kod dla każdego bajtu, przy użyciu podanych częstotliwości bajtów.
- const std::vector< uint8_t > * get_codes () const

  Zwraca listę kodów.
- bool try_get_byte (uint8_t &byte, uint8_t code_bit) const

  Przy użyciu statycznego bufora próbuje odczytać bajt z podanego kodu. Jeżeli kod nie jest jeszcze jednoznaczny funkcja dopisuje bit do bufora i zwraca false. W przeciwnym wypadku czyściu bufor, ustawia byte na odpowiedni bajt i zwraca true.

4.7.1 Opis szczegółowy

Reprezentacja drzewa Huffmana.

Definicja w linii 115 pliku huffman_tree.h.

4.7.2 Dokumentacja konstruktora i destruktora

4.7.2.1 huffman_tree()

```
huffman_tree::huffman_tree (
            const freq_map & chars_freq )
```

Tworzy drzewo Huffmana oraz generuje kod dla każdego bajtu, przy użyciu podanych częstotliwości bajtów.

Parametry

in	*chars_freq*	- struktura zawierająca częstotliwość bajtów

Definicja w linii 10 pliku huffman_tree.cpp.

4.7.2.2 ~huffman_tree()

```
huffman_tree::~huffman_tree ( )
```

Definicja w linii 81 pliku huffman_tree.cpp.

4.7.3 Dokumentacja funkcji składowych

4.7.3.1 get_codes()

```
const std::vector< uint8_t > * huffman_tree::get_codes ( ) const  [inline]
```

Zwraca listę kodów.

Zwraca

const std::vector<uint8_t>* wskaźnik do listy kodów

Definicja w linii 138 pliku huffman_tree.h.

4.7.3.2 try_get_byte()

```
bool huffman_tree::try_get_byte (
            uint8_t & byte,
            uint8_t code_bit ) const
```

Przy użyciu statycznego bufora próbuje odczytać bajt z podanego kodu. Jeżeli kod nie jest jeszcze jednoznaczny funkcja dopisuje bit do bufora i zwraca false. W przeciwnym wypadku czyściu bufor, ustawia byte na odpowiedni bajt i zwraca true.

Parametry

out	*byte*	bajt
	code_bit	bit kodu

Zwraca

true - jeżeli bajt został odczytany

false - jeżeli kod jest jeszcze niejednoznaczny

Definicja w linii 87 pliku huffman_tree.cpp.

Dokumentacja dla tej klasy została wygenerowana z plików:

- inc/huffman_tree.h
- src/huffman_tree.cpp

4.8 Dokumentacja klasy option

Klasa reprezentująca opcje wiersza linii poleceń

Metody publiczne

- option (std::string short_name, std::string long_name, std::string description, std::function< void(int &)> function)

  Tworzy nowy obiekt opcji.
- const std::string & get_short_name () const
- const std::string & get_long_name () const
- const std::string & get_description () const
- const std::function< void(int &)> & get_function () const
- bool matches (const std::string &candidate) const

  Funkcja sprawdzająca czy candidate jest równy short_name lub long_name.

4.8.1 Opis szczegółowy

Klasa reprezentująca opcje wiersza linii poleceń

Definicja w linii 27 pliku main.cpp.

4.8.2 Dokumentacja konstruktora i destruktora

4.8.2.1 option()

```
option::option (
            std::string short_name,
            std::string long_name,
            std::string description,
            std::function< void(int &)> function )  [inline]
```

Tworzy nowy obiekt opcji.

Parametry

short_name	- krótka nazwa
long_name	- długa nazwa
description	- opis
function	- akcja która zostanie wywołana kiedy przełącznik zostanie wykryty

Definicja w linii 45 pliku main.cpp.

4.8.3 Dokumentacja funkcji składowych

4.8.3.1 get_description()

```
const std::string & option::get_description ( ) const  [inline]
```

Definicja w linii 54 pliku main.cpp.

4.8.3.2 get_function()

```
const std::function< void(int &)> & option::get_function ( ) const  [inline]
```

Definicja w linii 55 pliku main.cpp.

4.8.3.3 get_long_name()

```
const std::string & option::get_long_name ( ) const  [inline]
```

Definicja w linii 53 pliku main.cpp.

4.8.3.4 get_short_name()

```
const std::string & option::get_short_name ( ) const  [inline]
```

Definicja w linii 52 pliku main.cpp.

4.8.3.5 matches()

```
bool option::matches (
            const std::string & candidate ) const  [inline]
```

Funkcja sprawdzająca czy candidate jest równy short_name lub long_name.

Parametry

candidate	- kandydat

Zwraca

true - kandydat odpowiada short_name lub long_name

false - jeżeli kandydat nie odpowiada short_name i long_name

Definicja w linii 68 pliku main.cpp.

Dokumentacja dla tej klasy została wygenerowana z pliku:

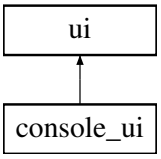
- src/main.cpp

4.9 Dokumentacja klasy ui

Prosty interfejs służący do komunikacji z urzytkownikiem.

```
#include <ui.h>
```

Diagram dziedziczenia dla ui



Metody publiczne

- virtual void write_message (const std::string &msg) const =0

  Wyświetla komunikat użytkownikowi.
- virtual void app_error (const std::string &error_msg) const =0

  Obsługuje błąd aplikacji.

4.9.1 Opis szczegółowy

Prosty interfejs służący do komunikacji z urzytkownikiem.

Definicja w linii 7 pliku ui.h.

4.9.2 Dokumentacja funkcji składowych

4.9.2.1 app_error()

```
virtual void ui::app_error (
            const std::string & error_msg ) const  [pure virtual]
```

Obsługuje błąd aplikacji.

Parametry

error_msg	- komunikat błędu

Implementowany w console_ui.

4.9.2.2 write_message()

```
virtual void ui::write_message (
            const std::string & msg ) const  [pure virtual]
```

Wyświetla komunikat użytkownikowi.

Parametry

msg	- komunikat

Implementowany w console_ui.

Dokumentacja dla tej klasy została wygenerowana z pliku:

- inc/ui.h

Rozdział 5

Dokumentacja plików

5.1 bit_file_io.h

```
00001 #pragma once
00002
00003 #include <cstdint>
00004 #include <fstream>
00005
00010 class bit_file_io
00011 {
00012   private:
00013     std::fstream &file_stream_;
00014     // buffers for bit manipulation
00015     uint8_t r_bit_buf_ = 0, r_bit_buf_size_ = 0;
00016     uint8_t w_bit_buf_ = 0, w_bit_buf_size_ = 0;
00017
00018     // buffers for reading/writing from/to file
00019     uint8_t *r_buff_;
00020     size_t r_buff_cnt_ = 0, r_buff_size_;
00021     uint8_t *w_buff_;
00022     size_t w_buff_cnt_ = 0, w_buff_size_;
00023
00024   public:
00025     bit_file_io(std::fstream &file_stream, size_t read_buff_size,
00026                 size_t write_buff_size);
00027     ~bit_file_io();
00028
00033     void write_bit(uint8_t bit);
00034
00038     void flush_bit_buffer();
00039
00043     void flush_buffer();
00044
00051     bool read_bit(uint8_t &bit);
00052
00053     bit_file_io &operator«(const uint8_t bit);
00054     bool operator»(uint8_t &bit);
00055 };
```

5.2 consts.h

```
00001 #pragma once
00002
00003 #include <cstddef>
00004
00008 #define UNUSED(x) (void)x
00009
00013 static constexpr size_t size_16_mb = 16777216; // 16mb
00014
00018 static constexpr size_t size_8_mb = 8388608; // 8mb
```

5.3 huffman_encoder.h

```
00001 #pragma once
00002
00003 #include <cstdint>
00004 #include <string>
00005
00006 #include "consts.h"
00007 #include "ui.h"
00008
00012 class huffman_encoder
00013 {
00014   private:
00015     std::string input_file_, output_file_;
00016     const ui &ui_;
00017
00018     uint8_t *const buffer_;
00019     const size_t buffer_size_;
00020     size_t buffer_cnt_ = 0;
00021
00022   public:
00031     huffman_encoder(std::string input_file, std::string output_file,
00032                     const ui &ui, const size_t buffer_size = size_16_mb);
00033     ~huffman_encoder();
00034
00038     void compress_file();
00042     void decompress_file();
00043 };
```

5.4 huffman_tree.h

```
00001 #pragma once
00002 #include <cstdint>
00003 #include <unordered_map>
00004 #include <vector>
00005
00009 class freq_map
00010 {
00011   private:
00012     std::vector<uint64_t> freq_;
00013
00014   public:
00015     freq_map() : freq_(std::vector<uint64_t>(UINT8_MAX + 1, 0)) {}
00016
00023     uint64_t get(uint8_t byte) const { return freq_[byte]; }
00024
00031     void set(uint8_t byte, uint64_t value) { freq_[byte] = value; }
00032
00038     void inc(uint8_t byte) { ++freq_[byte]; }
00039
00044     uint16_t size() const
00045     {
00046         uint16_t cnt = 0;
00047         for (uint16_t i = 0; i <= UINT8_MAX; i++)
00048             if (freq_[i])
00049                 cnt++;
00050         return cnt;
00051     };
00052 };
00053
00057 class huffman_node
00058 {
00059   private:
00060     uint64_t freq_ = 0;
00061     huffman_node *left_node_{}, *right_node_{};
00062
00063   public:
00064     huffman_node(const uint64_t frequency, huffman_node *left_child,
00065                  huffman_node *right_child)
00066         : freq_(frequency), left_node_(left_child), right_node_(right_child)
00067     {
00068     }
00069     virtual ~huffman_node() = default;
00070
00075     uint64_t get_frequency() const { return this->freq_; }
00076
00081     huffman_node *get_left_child() const { return this->left_node_; }
00082
00087     huffman_node *get_right_child() const { return this->right_node_; }
00088 };
00089
00094 class huffman_leaf : public huffman_node
00095 {
```

```
00096   private:
00097     uint8_t value_ = 0;
00098
00099   public:
00100     huffman_leaf(const uint64_t frequency, const uint8_t value)
00101         : huffman_node(frequency, nullptr, nullptr), value_(value)
00102     {
00103     }
00104
00109     uint8_t get_value() const { return this->value_; }
00110 };
00111
00115 class huffman_tree
00116 {
00117   private:
00118     huffman_node *tree_root_ = nullptr;
00119     const freq_map &chars_freq_;
00120     std::vector<uint8_t> *codes_;
00121     void fill_codes(huffman_node *root, std::vector<uint8_t> current);
00122
00123   public:
00130     huffman_tree(const freq_map &chars_freq);
00131     ~huffman_tree();
00132
00138     const std::vector<uint8_t> *get_codes() const { return this->codes_; }
00139
00151     bool try_get_byte(uint8_t &byte, uint8_t code_bit) const;
00152 };
```

5.5 ui.h

```
00001 #pragma once
00002 #include <string>
00003
00007 class ui
00008 {
00009   public:
00010     virtual ~ui() = default;
00015     virtual void write_message(const std::string &msg) const = 0;
00016
00021     virtual void app_error(const std::string &error_msg) const = 0;
00022 };
00023
00028 class console_ui final : public ui
00029 {
00030   public:
00035     void write_message(const std::string &msg) const override;
00036
00042     void app_error(const std::string &error_msg) const override;
00043 };
```

5.6 bit_file_io.cpp

```
00001 #include "bit_file_io.h"
00002
00003 #include <climits>
00004
00005 bit_file_io::bit_file_io(std::fstream &file_stream, size_t read_buff_size,
00006                          size_t write_buff_size)
00007     : file_stream_(file_stream),
00008       r_buff_size_(std::min(read_buff_size, static_cast<size_t>(1))),
00009       w_buff_size_(std::min(write_buff_size, static_cast<size_t>(1)))
00010 {
00011     r_buff_ = new uint8_t[r_buff_size_];
00012     w_buff_ = new uint8_t[w_buff_size_];
00013 }
00014
00015 bit_file_io::~bit_file_io()
00016 {
00017     delete r_buff_;
00018     delete w_buff_;
00019 }
00020
00021 void bit_file_io::write_bit(uint8_t bit)
00022 {
00023     bit &= 1;
00024
00025     if (bit)
00026         this->w_bit_buf_ |= 1 « ((CHAR_BIT - 1) - this->w_bit_buf_size_);
```

```
00027     this->w_bit_buf_size_++;
00028
00029     if (this->w_bit_buf_size_ == CHAR_BIT)
00030         flush_bit_buffer();
00031 }
00032
00037 void bit_file_io::flush_bit_buffer()
00038 {
00039     if (this->w_bit_buf_size_ == 0)
00040         return;
00041
00042     this->w_buff_[this->w_buff_cnt_++] = this->w_bit_buf_;
00043     this->w_bit_buf_size_ = 0;
00044     this->w_bit_buf_ = 0;
00045
00046     if (this->w_buff_cnt_ == this->w_buff_size_)
00047         flush_buffer();
00048 }
00049
00054 void bit_file_io::flush_buffer()
00055 {
00056     if (this->w_buff_cnt_ == 0)
00057         return;
00058
00059     file_stream_.write(reinterpret_cast<char *>(this->w_buff_),
00060                        sizeof(uint8_t) * this->w_buff_cnt_);
00061     this->w_buff_cnt_ = 0;
00062 }
00063
00064 bool bit_file_io::read_bit(uint8_t &bit)
00065 {
00066     if (this->r_bit_buf_size_ == 0)
00067     {
00068         if (this->r_buff_cnt_ == 0)
00069         {
00070             if (!this->file_stream_.read(
00071                     reinterpret_cast<char *>(this->r_buff_),
00072                     this->r_buff_size_))
00073                 return false;
00074             this->r_buff_cnt_ = this->file_stream_.gcount();
00075         }
00076
00077         this->r_bit_buf_ = this->r_buff_[--r_buff_cnt_];
00078         this->r_bit_buf_size_ = CHAR_BIT;
00079     }
00080
00081     this->r_bit_buf_size_--;
00082     bit = r_bit_buf_ & (1 « this->r_bit_buf_size_) » this->r_bit_buf_size_;
00083     return true;
00084 }
00085
00086 bit_file_io &bit_file_io::operator«(const uint8_t bit)
00087 {
00088     this->write_bit(bit);
00089     return *this;
00090 }
00091
00092 bool bit_file_io::operator»(uint8_t &bit) { return this->read_bit(bit); }
```

5.7 huffman_encoder.cpp

```
00001 #include "huffman_encoder.h"
00002
00003 #include "bit_file_io.h"
00004 #include "huffman_tree.h"
00005 #include "ui.h"
00006 #include <algorithm>
00007 #include <climits>
00008 #include <fstream>
00009 #include <iostream>
00010 #include <unordered_map>
00011
00012 static void write_file_header(const freq_map &map, uint8_t padding,
00013                               std::fstream &output_file, bit_file_io &wrapper);
00014 static huffman_tree *read_file_header(std::fstream &file, bit_file_io &wrapper);
00015
00016 huffman_encoder::huffman_encoder(std::string input_file,
00017                                  std::string output_file, const ui &ui,
00018                                  const size_t buffer_size)
00019     : input_file_(std::move(input_file)), output_file_(std::move(output_file)),
00020       ui_(ui), buffer_(new uint8_t[buffer_size]), buffer_size_(buffer_size)
00021 {
00022 }
```

```
00023
00024 huffman_encoder::huffman_encoder(const huffman_encoder &cpy)
00025     : input_file_(cpy.input_file_), output_file_(cpy.output_file_),
00026       ui_(cpy.ui_), buffer_(new uint8_t[cpy.buffer_size_]),
00027       buffer_size_(cpy.buffer_size_)
00028 {
00029 }
00030
00031 huffman_encoder::~huffman_encoder() { delete[] buffer_; }
00032
00033 void huffman_encoder::compress_file()
00034 {
00035     this->ui_.write_message("Starting compression...");
00036     this->ui_.write_message("Output file: " + this->output_file_);
00037     std::fstream input_file(this->input_file_,
00038                             std::ios::in | std::ios_base::binary);
00039
00040     if (!input_file.good() ||
00041         input_file.peek() == std::ifstream::traits_type::eof())
00042     {
00043         input_file.close();
00044         this->ui_.app_error("Input file doesn't exists, or it's empty.");
00045         return;
00046     }
00047
00048     std::fstream output_file(this->output_file_,
00049                              std::ios::out | std::ios_base::binary);
00050     if (!output_file.good())
00051     {
00052         input_file.close();
00053         output_file.close();
00054         this->ui_.app_error("Cannot create or write to output file.");
00055         return;
00056     }
00057
00058     bit_file_io output_file_bit_io(output_file, 1, size_16_mb);
00059
00060     this->ui_.write_message("Counting byte frequency...");
00061
00062     freq_map map;
00063     // https://stackoverflow.com/a/67854635
00064
00065     while (input_file.good())
00066     {
00067         input_file.read(
00068             reinterpret_cast<char *>(this->buffer_),
00069             static_cast<std::streamsize>(sizeof(uint8_t) * this->buffer_size_));
00070         this->buffer_cnt_ = static_cast<size_t>(input_file.gcount());
00071         for (size_t i = 0; i < buffer_cnt_; i++)
00072         {
00073             uint8_t byte = this->buffer_[i];
00074             map.inc(byte);
00075         }
00076     }
00077
00078     this->ui_.write_message("Finished counting bytes.");
00079
00080     this->ui_.write_message("Building huffman tree...");
00081     const huffman_tree *tree = new huffman_tree(map);
00082     auto codes = tree->get_codes();
00083     this->ui_.write_message("Tree created, and codes generated.");
00084
00085     // calculate needed padding
00086     uint8_t padding = 0;
00087     for (uint16_t i = 0; i <= UINT8_MAX; i++)
00088         padding = (padding + ((codes[i].size() % CHAR_BIT) *
00089                               (map.get(static_cast<uint8_t>(i)) % CHAR_BIT)) %
00090                                  CHAR_BIT) %
00091                   CHAR_BIT;
00092
00093     this->ui_.write_message("Writing file header...");
00094     write_file_header(map, CHAR_BIT - padding, output_file, output_file_bit_io);
00095     this->ui_.write_message("File header written.");
00096
00097     input_file.clear(); // clear eof flag
00098     input_file.seekg(0, std::ios_base::beg);
00099
00100     this->ui_.write_message("Encoding bytes...");
00101     while (input_file.good())
00102     {
00103         input_file.read(
00104             reinterpret_cast<char *>(this->buffer_),
00105             static_cast<std::streamsize>(sizeof(uint8_t) * this->buffer_size_));
00106         this->buffer_cnt_ = static_cast<size_t>(input_file.gcount());
00107         for (size_t i = 0; i < this->buffer_cnt_; i++)
00108         {
00109             uint8_t byte = this->buffer_[i];
```

```
00110             auto &code = codes[byte];
00111             for (bool bit : code)
00112                 output_file_bit_io « (bit ? 1 : 0);
00113         }
00114     }
00115
00116     output_file_bit_io.flush_bit_buffer();
00117     output_file_bit_io.flush_buffer();
00118
00119     this->ui_.write_message("Compression finished");
00120
00121     input_file.close();
00122     output_file.close();
00123
00124     delete tree;
00125 }
00126
00127 void huffman_encoder::decompress_file()
00128 {
00129     this->ui_.write_message("Starting decompression...");
00130
00131     std::fstream input_file(this->input_file_,
00132                             std::ios::in | std::ios_base::binary);
00133     if (!input_file.good() ||
00134         input_file.peek() == std::ifstream::traits_type::eof())
00135     {
00136         input_file.close();
00137         this->ui_.app_error("Input file doesn't exists, or it's empty.");
00138         return;
00139     }
00140     bit_file_io input_file_bit_io(input_file, size_16_mb, 1);
00141
00142     std::fstream output_file(this->output_file_,
00143                              std::ios::out | std::ios_base::binary);
00144     if (!output_file.good())
00145     {
00146         input_file.close();
00147         output_file.close();
00148         this->ui_.app_error("Cannot create or write to output file.");
00149         return;
00150     }
00151
00152     this->ui_.write_message("Reading file header and rebuilding tree...");
00153     const huffman_tree *tree;
00154     try
00155     {
00156         tree = read_file_header(input_file, input_file_bit_io);
00157     }
00158     catch (const std::logic_error &ex)
00159     {
00160         ui_.app_error(ex.what());
00161         return;
00162     }
00163     this->ui_.write_message("Tree created.");
00164
00165     this->ui_.write_message("Transforming bytes...");
00166     uint8_t bit = 0, byte = 0;
00167     while (input_file_bit_io » byte)
00168     {
00169         if (tree->try_get_byte(byte, bit))
00170         {
00171             this->buffer_[this->buffer_cnt_++] = byte;
00172             if (this->buffer_cnt_ == this->buffer_size_)
00173             {
00174                 output_file.write(reinterpret_cast<char *>(this->buffer_),
00175                                   static_cast<std::streamsize>(
00176                                       sizeof(uint8_t) * this->buffer_size_));
00177                 this->buffer_cnt_ = 0;
00178             }
00179         }
00180     }
00181
00182     if (buffer_cnt_ > 0)
00183     {
00184         output_file.write(
00185             reinterpret_cast<char *>(this->buffer_),
00186             static_cast<std::streamsize>(sizeof(uint8_t) * this->buffer_cnt_));
00187         this->buffer_cnt_ = 0;
00188     }
00189
00190     this->ui_.write_message("Decompression finished");
00191     input_file.close();
00192     output_file.close();
00193
00194     delete tree;
00195 }
00196
```

```
00197 static void write_file_header(const freq_map &map, const uint8_t padding,
00198                                std::fstream &output_file, bit_file_io &wrapper)
00199 {
00200     uint8_t buf[2] = {
00201         static_cast<uint8_t>(map.size() -
00202                              1), // bytes_size + 1 = unique bytes count;
00203         padding                 // needed padding
00204     };
00205
00206     output_file.write(reinterpret_cast<char *>(&buf), sizeof(buf));
00207
00208     for (uint16_t chr = 0; chr <= UINT8_MAX; chr++)
00209     {
00210         if (uint64_t frq = map.get(static_cast<uint8_t>(chr)))
00211         {
00212             output_file.write(reinterpret_cast<char *>(&chr), sizeof(uint8_t));
00213             output_file.write(reinterpret_cast<char *>(&frq), sizeof(uint64_t));
00214         }
00215     }
00216
00217     for (uint8_t i = 0; i < padding; i++)
00218         wrapper « 0;
00219     // no need to flush it here
00220     // flushing will cause problems
00221 }
00222
00223 static huffman_tree *read_file_header(std::fstream &file, bit_file_io &wrapper)
00224 {
00225     uint8_t header[2];
00226     // header[0] -> num of unique bytes - 1
00227     // header[1] -> padding at the end
00228     file.read(reinterpret_cast<char *>(&header), sizeof(header));
00229     const uint16_t unique_bytes = static_cast<uint16_t>(header[0]) + 1;
00230     freq_map map;
00231     uint8_t byte = 0;
00232     uint64_t count = 0;
00233     uint16_t bytes_read = 2;
00234
00235     while (bytes_read < ((unique_bytes * 9) + 2) &&
00236            file.read(reinterpret_cast<char *>(&byte), sizeof(uint8_t)) &&
00237            file.read(reinterpret_cast<char *>(&count), sizeof(uint64_t)))
00238     {
00239         bytes_read += 9;
00240         map.set(byte, count);
00241     }
00242
00243     uint8_t bit;
00244     for (uint8_t i = 0; i < header[1] && wrapper » bit; i++)
00245         ;
00246
00247     return new huffman_tree(map);
00248 }
```

5.8 huffman_tree.cpp

```
00001 #include "huffman_tree.h"
00002
00003 #include <queue>
00004 #include <stdexcept>
00005 #include <utility>
00006
00007 static void delete_tree(const huffman_node *root);
00008 static huffman_leaf *get_left_most_leaf(huffman_node *node);
00009
00010 huffman_tree::huffman_tree(const freq_map &chars_freq) : chars_freq_(chars_freq)
00011 {
00012     // https://stackoverflow.com/a/5808171
00013     // min heap comparator
00014     auto comp = [](huffman_node *n1, huffman_node *n2)
00015     {
00016         const uint64_t freq1 = n1->get_frequency();
00017         const uint64_t freq2 = n2->get_frequency();
00018
00019         // if booth of nodes have equal frequency and are leafs, choose one with
00020         // smaller value
00021         if (freq1 == freq2)
00022         {
00023             const huffman_leaf *leaf1 = dynamic_cast<huffman_leaf *>(n1),
00024                                *leaf2 = dynamic_cast<huffman_leaf *>(n2);
00025             if (leaf1 != nullptr && leaf2 != nullptr)
00026             {
00027                 return leaf1->get_value() > leaf2->get_value();
00028             }
```

```
00029             if (leaf1 != nullptr)
00030             {
00031                 return false;
00032             }
00033             if (leaf2 != nullptr)
00034             {
00035                 return true;
00036             }
00037             leaf1 = get_left_most_leaf(n1);
00038             leaf2 = get_left_most_leaf(n2);
00039             return leaf1->get_value() > leaf2->get_value();
00040         }
00041         return freq1 > freq2;
00042     };
00043     std::priority_queue<huffman_node *, std::vector<huffman_node *>,
00044                         decltype(comp)>
00045         pq(comp);
00046
00047     for (uint16_t chr = 0; chr <= UINT8_MAX; chr++)
00048         if (const uint64_t freq =
00049                 this->chars_freq_.get(static_cast<uint8_t>(chr)))
00050             pq.push(new huffman_leaf(freq, static_cast<uint8_t>(chr)));
00051
00052     if (pq.empty())
00053         throw std::logic_error("Cannot create tree with 0 unique bytes.");
00054
00055     if (pq.size() == 1)
00056     {
00057         const auto node = pq.top();
00058         pq.pop();
00059         pq.push(new huffman_node(node->get_frequency(), node, nullptr));
00060     }
00061
00062     while (pq.size() > 1)
00063     {
00064         auto node1 = pq.top();
00065         pq.pop();
00066         auto node2 = pq.top();
00067         pq.pop();
00068
00069         auto parent = new huffman_node(
00070             node1->get_frequency() + node2->get_frequency(), node1, node2);
00071         pq.push(parent);
00072     }
00073
00074     tree_root_ = pq.top();
00075     pq.pop();
00076
00077     codes_ = new std::vector<uint8_t>[UINT8_MAX + 1];
00078     fill_codes(this->tree_root_, {});
00079 }
00080
00081 huffman_tree::~huffman_tree()
00082 {
00083     delete_tree(this->tree_root_);
00084     delete[] codes_;
00085 }
00086
00087 bool huffman_tree::try_get_byte(uint8_t &byte, uint8_t code_bit) const
00088 {
00089     static huffman_node *current_node = this->tree_root_;
00090     if (code_bit)
00091         current_node = current_node->get_right_child();
00092     else
00093         current_node = current_node->get_left_child();
00094
00095     if (const auto leaf = dynamic_cast<huffman_leaf *>(current_node))
00096     {
00097         byte = leaf->get_value();
00098         current_node = this->tree_root_;
00099         return true;
00100     }
00101     return false;
00102 }
00103
00104 void huffman_tree::fill_codes(huffman_node *root, std::vector<uint8_t> current)
00105 {
00106     if (root == nullptr)
00107         return;
00108
00109     if (const auto leaf = dynamic_cast<huffman_leaf *>(root))
00110     {
00111         this->codes_[leaf->get_value()] = std::move(current);
00112         return;
00113     }
00114
00115     auto cpy = std::vector<uint8_t>(current);
```

```
00116     current.push_back(0);
00117     fill_codes(root->get_left_child(), current);
00118     cpy.push_back(1);
00119     fill_codes(root->get_right_child(), cpy);
00120 }
00121
00122 static void delete_tree(const huffman_node *root)
00123 {
00124     if (root == nullptr)
00125         return;
00126     if (root->get_left_child() != nullptr)
00127         delete_tree(root->get_left_child());
00128     if (root->get_right_child() != nullptr)
00129         delete_tree(root->get_right_child());
00130     delete root;
00131 }
00132
00133 static huffman_leaf *get_left_most_leaf(huffman_node *node)
00134 {
00135     if (node == nullptr)
00136     {
00137         throw std::logic_error("Unknown error. Cannot create tree");
00138     }
00139
00140     huffman_leaf *leaf;
00141     while ((leaf = dynamic_cast<huffman_leaf *>(node)) == nullptr)
00142     {
00143         node = node->get_left_child();
00144     }
00145     return leaf;
00146 }
```

5.9 main.cpp

```
00001 #include <cstdlib>
00002 #include <functional>
00003 #include <iostream>
00004 #include <sstream>
00005 #include <vector>
00006
00007 #include "consts.h"
00008 #include "huffman_encoder.h"
00009 #include "huffman_tree.h"
00010 #include "ui.h"
00011
00012 static const console_ui console_ui;
00013
00014 static const std::string mode_compress = "compress";
00015 static const std::string mode_decompress = "decompress";
00016
00017 enum class mode
00018 {
00019     INVALID = 0,
00020     COMPRESS,
00021     DECOMPRESS,
00022 };
00023
00027 class option
00028 {
00029   private:
00030     const std::string short_name_, long_name_;
00031     const std::string description_;
00032
00033     const std::function<void(int &)> function_;
00034
00035   public:
00045     option(std::string short_name, std::string long_name,
00046            std::string description, std::function<void(int &)> function)
00047         : short_name_(std::move(short_name)), long_name_(std::move(long_name)),
00048           description_(std::move(description)), function_(std::move(function))
00049     {
00050     }
00051
00052     const std::string &get_short_name() const { return this->short_name_; }
00053     const std::string &get_long_name() const { return this->long_name_; }
00054     const std::string &get_description() const { return this->description_; }
00055     const std::function<void(int &)> &get_function() const
00056     {
00057         return this->function_;
00058     }
00059
00068     bool matches(const std::string &candidate) const
00069     {
```

```
00070         return this->get_long_name() == candidate ||
00071                this->get_short_name() == candidate;
00072     }
00073 };
00074
00075 static void invalid_usage(const std::string &program_name)
00076 {
00077     std::stringstream ss;
00078     ss « "Invalid arguments. Run: " « program_name « " --help for help";
00079     console_ui.app_error(ss.str());
00080 }
00081
00082 int main(int argc, char *argv[])
00083 {
00084     try
00085     {
00086         const std::string program_name = argv[0];
00087         std::string input_file, output_file;
00088         auto mode = mode::INVALID;
00089
00090         std::vector<option> options{
00091             option("-h", "--help", "Prints help",
00092                    [program_name, &options](int &i)
00093                    {
00094                        UNUSED(i);
00095                        std::stringstream ss;
00096
00097                        ss « "Running: " « program_name « " [OPTIONS]";
00098                        console_ui.write_message(ss.str());
00099                        // https://stackoverflow.com/a/20792
00100                        ss.str(std::string());
00101
00102                        console_ui.write_message("Options: ");
00103
00104                        for (const auto &option : options)
00105                        {
00106                            ss « "\t" « option.get_short_name() « ", "
00107                               « option.get_long_name() « " - "
00108                               « option.get_description();
00109                            console_ui.write_message(ss.str());
00110                            ss.str(std::string());
00111                        }
00112                        exit(EXIT_SUCCESS);
00113                    }),
00114             option("-i", "--input-file", "Input file path [required]",
00115                    [argc, argv, &input_file](int &i)
00116                    {
00117                        if (i + 1 >= argc)
00118                            console_ui.app_error("Input file not specified");
00119                        input_file = std::string(argv[i + 1]);
00120                        i++;
00121                    }),
00122             option(
00123                 "-o", "--output-file",
00124                 "Output file path [optional, defaults to $(input-file).out)]",
00125                 [argc, argv, &output_file](int &i)
00126                 {
00127                     if (i + 1 >= argc)
00128                         console_ui.app_error("Output file not specified");
00129                     output_file = std::string(argv[i + 1]);
00130                     i++;
00131                 }),
00132             option("-m", "--mode",
00133                    "Compression algorithm mode <" + mode_compress + "|" +
00134                        mode_decompress + "> [required]",
00135                    [argc, argv, &mode](int &i)
00136                    {
00137                        if (i + 1 >= argc)
00138                            console_ui.app_error("Mode not specified");
00139                        if (argv[i + 1] == mode_compress)
00140                            mode = mode::COMPRESS;
00141                        else if (argv[i + 1] == mode_decompress)
00142                            mode = mode::DECOMPRESS;
00143                        i++;
00144                    })};
00145
00146         if (argc < 2)
00147             invalid_usage(program_name);
00148
00149         for (int i = 1; i < argc; i++)
00150             for (const auto &option : options)
00151                 if (option.matches(argv[i]))
00152                     option.get_function()(i);
00153
00154         if (mode == mode::INVALID)
00155             invalid_usage(program_name);
00156
```

```
00157         if (input_file.empty())
00158             invalid_usage(program_name);
00159
00160         if (output_file.empty())
00161             output_file = input_file + ".out";
00162
00163         auto encoder = huffman_encoder(input_file, output_file, console_ui);
00164
00165         switch (mode)
00166         {
00167         case mode::COMPRESS:
00168             encoder.compress_file();
00169             break;
00170         case mode::DECOMPRESS:
00171             encoder.decompress_file();
00172             break;
00173
00174         case mode::INVALID:
00175         default:
00176             console_ui.app_error("Unhandled mode.");
00177             break;
00178         }
00179
00180         return EXIT_SUCCESS;
00181     }
00182     catch (const std::exception &ex)
00183     {
00184         std::cerr << ex.what();
00185         return EXIT_FAILURE;
00186     }
00187 }
```

5.10 ui.cpp

```
00001 #include "ui.h"
00002
00003 #include <iostream>
00004
00005 void console_ui::write_message(const std::string& msg) const
00006 {
00007     std::cout << msg << std::endl;
00008 }
00009
00010 void console_ui::app_error(const std::string& error_msg) const
00011 {
00012     std::cerr << error_msg << std::endl;
00013     exit(EXIT_FAILURE);
00014 }
```

Skorowidz