Міністерство освіти і науки України Національний університет „Львівська політехніка”

Кафедра ЕОМ



**Курсова робота**

з курсу:

„Системне програмне забезпечення”

на тему:

**„ Розробка утиліти стискання файлів за алгоритмом LZW ”**

Виконав:

студент групи КІ-37

Маркович О.І

Перевірив:

Жолубак І.М

Львів 2023

**АНОТАЦІЯ**

В курсовій роботі здійснена розробка утиліти стискання файлів з використанням алгоритму LZW. За допомогою цієї програми можна стискати інформацію за цим алгоритмом. Після стиснення утворюється файл з розширенням .LZW, його можна розархівувати скориставшись цією ж програмою.

Курсова робота виконана у середовищі Microsoft Visual Studio 2019, лістинг програми наведений у додатку А. Також у курсовій роботі наведено граф-схеми роботи програми, алгоритм архівування і розархівування, детальний опис програми з поясненням усіх використовуваних функцій і змінних.

До проекту додано результати тестування програми та опис інтерфейсу та інструкції користувачеві.

Зміст

**Вступ**

Разом із виникненням комп’ютерів постала проблема збереження інформації на ньому. Саме тому і виникло багато алгоритмів стиснення інформації і програм які їх виконують.

Архіватори - це програми що дозволяють створювати і обробляти архівні копії файлів. При цьому архівні копії мають менший розмір ніж оригінали. За допомогою спеціальних алгоритмів стиску з файлів віддаляється вся надлишкова інформація, а при застосуванні зворотніх алгоритмів розпакування - архівна копія відновлюєть в первісному виді.

Безумовним лідером у світі за останні п’ять років став архіватор RAR. В даний час RAR активно витісняє ZIP як основну утиліту стиску FTP архівів у мережі INTERNET. RAR є єдиною всесвітньо-відомою програмою створеною російським програмістом( за винятком TETRIS). Всі архіватори відрізняються використовуваними алгоритмами стиску, форматами архівних файлів, швидкістю роботи тощо.

Одним із поширених алгоритмів стиску інформації є алгоритм LZW(Lempel-Ziv-Welch), він використовується PDF i TIFF форматах файлів. Цей алгоритм стиснення інформації є темою моєї курсової роботи.

* Вибір технології програмування

Цільовою мовою програмування було обрано C++. Дана мова програмування широко використовується для розв’язання задач різного рівня складності. Перевагами даної мови порівняно з іншими є широкий набір бібліотек класів для роботи з стрічками, файлами, різноманітними структурами даних.

Оскільки утиліта орієнтована на сімейство операційних систем Windows, то для її розробки будуть використовуватись Windows API функції - набір базових функцій інтерфейсів програмування. Вони дадуть змогу просто і зручно отримувати інформацію про потрібні нам параметри. Нижче буде наведено основні функції які нам потрібні для роботи та їх опис.

Технологією програмування було обране процедурне програмування.

Виконання програми зводиться до послідовного виконання операторів з метою перетворення початкового стану пам'яті, тобто значень вихідних даних, в заключне, тобто в результати. Таким чином, з точки зору програміста є програма і пам'ять, причому перша послідовно оновлює вміст останньої.

Процедурна мова програмування надає можливість програмісту визначати кожен крок у процесі рішення задачі. Особливість таких мов програмування полягає в тому, що завдання розбиваються на кроки і вирішуються крок за кроком. Використовуючи процедурну мову, програміст визначає мовні конструкції для виконання послідовності алгоритмічних кроків.

* Огляд алгоритму архівації

Групове кодування Run Lengh Encodind(LZW) –один із самих простих алгоритмів архівації. Стискання LZW відбувається за рахунок заміни однакових байт на пари «лічильник, значення».

Незважаючи на давнє народження LZW як алгоритму стиску зображення і на його сьогоднішню неефективність у світі складності (наявність дрібних деталей) сучасних зображень, всетаки сьогодні існує клас програм, де використання LZW більш виправдано ніж використання інших алгоритмів. Позитивним у цьому алгоритмі є його простота, а значить це зменшує фінансові(між іншим і апаратні) витрати на використання блоків деякої контролюючої системи. Алгоритм застосовується у форматах TIFF, PDF.

Ось наведено приклад використання даного алгоритму:

Даний приклад показує алгоритм LZW в дії, зображуючи стан вихідних даних і словника на кожній стадії, як при кодуванні, так і при розкодуванні повідомлення. Щоб зробити вкладення простішим, ми обмежимося простим алфавітом — тільки прописні букви, без знаків пунктуації і пробілів. Повідомлення, яке потрібно стиснути, виглядає наступним чином:

Маркер **#** використовується для позначення кінця повідомлення. Отже в нашому алфавіті 27 символів (26 прописних букв від A до Z і #). Комп'ютер представляє це у вигляді груп бітів, для представлення кожного символу алфавіту нам достатньо групи з 5 бітів на символ. По мірі росту словника розмір груп повинен рости, щоб врахувати нові елементи. 5-бітні групи дають 25 = 32 можливих комбінацій бітів, тому, коли в словнику з'явиться 33-тє слово, алгоритм повинен перейти до 6-бітних груп. Зауважимо, що, оскільки використовується група з всіх нулів 00000, то 33-тя група має код **32**. Початковий словник міститиме:

|  |  |
| --- | --- |
| # = | 00000 |
| A = | 00001 |
| B = | 00010 |
| C = | 00011 |
| . |  |
| . |  |
| . |  |
| Z = | 11010 |
| Без | використання алгоритму LZW, при передачі повідомлення у початковому |

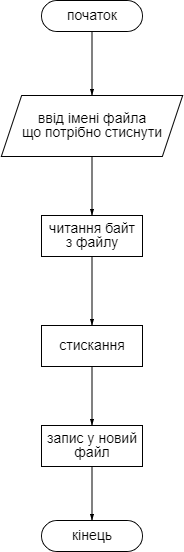
вигляді — 25 символів по 5 бітів на кожен символ — воно займає 125 бітів. Порівняємо це з тим, що отримуємо при використанні LZW:

Таким чином, використовуючи LZW ми скоротили повідомлення на 29 бітів з 125 — це майже 22 %. Якщо повідомлення буде довшим, то елементи словника будуть представляти все більш і більш довгі частини тексту, завдяки чому слова, що повторюються, будуть представлені дуже компактно.

* Розробка утиліти стискання методом LZW

Для ефективної роботи створюваної програми важливу роль відіграє попереднє складення алгоритму роботи програми, алгоритму написання програми і вибір технології програмування. Дана програма написана мовою C++.

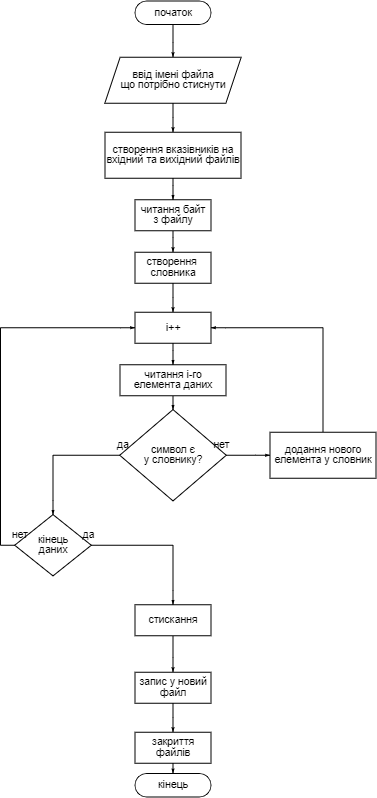
Граф схема алгоритму роботи програми наведена нижче



*Рис.1 Граф-схема алгоритму програми*

На рисунку наведена граф-схема алгоритму роботи програми 1- початок роботи програми. Алгоритм стиснення реалізований функцією void compress()

Далі наведено детальну схему роботи програми, а саме граф-схема роботи стиснення файлів і роботи розтиснення файлів.



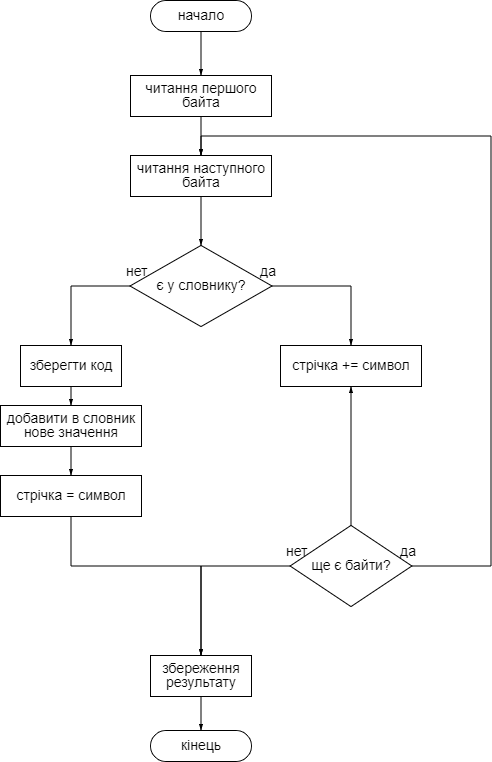
*Рис.2 Граф-схема роботи стиснення файлів*

На рисунку 2 наведено схему роботи стиснення файлів.початок роботи функції void compress.

створення LZW файлу, воно відбувається за допомогою функції std::ofstream.

визначення кількості байт вхідного файлу за допомогою функції size(). Реалізація алгоритму, якщо зустрічаються однакові байти то виконується функція search\_and\_insert();

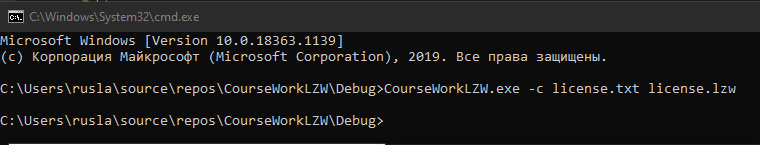
якщо пара нова – записує її в словник, в іншому випадку продовжує роботу, після проходження всього файлу виконується запис стисненої інформації у файл і завершення роботи функції.



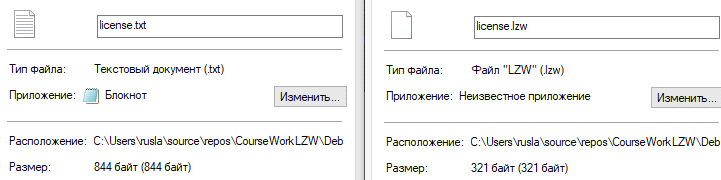
*Рис.2 Граф-схема роботи розархівування файлів*

* Опис інтерфейсу та інструкції користувачеві

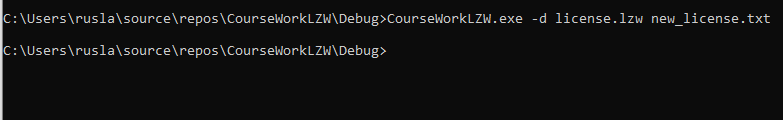
Створена утиліта написана мовою C++. Архіватор складається з поля де потрібно вписати назву файлу який потрібно стиснути і назву нового (стиснутого) файлу.

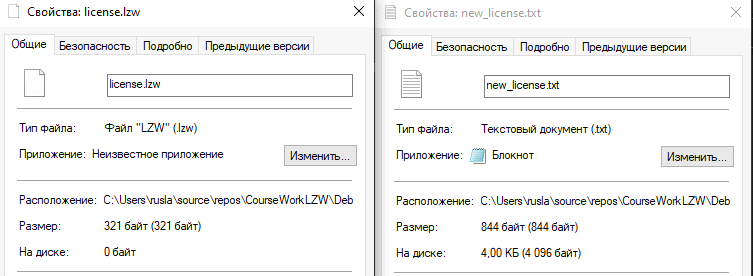


При натисненні клавіші enter файл архівується.



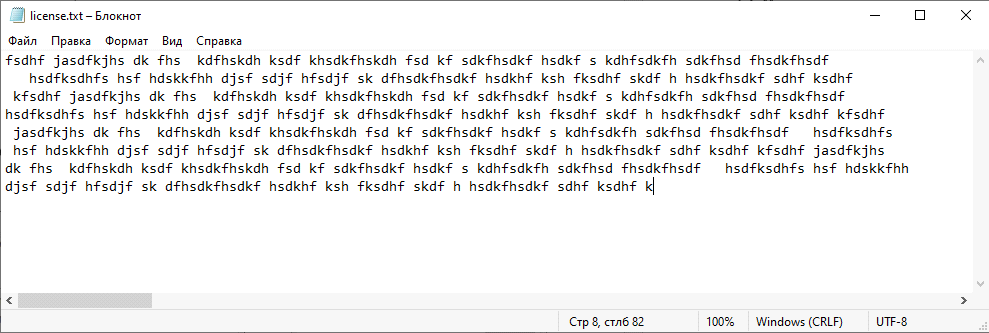
Aрхівований файл буде мати розширення .lzw. Для того щоб розархівувати файл потрібно під час виклику програми вказати тип операції decompress(-d) та дати назву розархівованому файлу





* Тестування програми

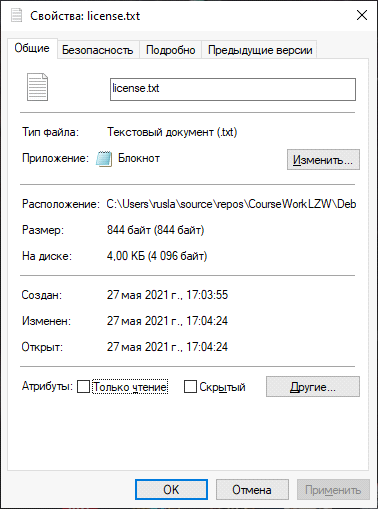
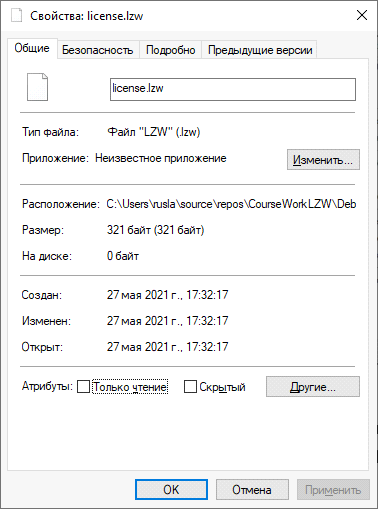
Щоб перевірити правильність роботи алгоритму проведено такі тестування: Спочатку перевіримо ввівши у текстовий файл такі дані:



Перед стисненням його розмір

був:

Після стиснення став:

Після розархівування інформація залишилась незмінною.

Висновок

В процесі виконання курсової роботи було виконано наступне: Розроблено утиліту стискання файлів за алгоритмом LZW мовою C++. Наведено граф-схема загальної роботи програми

Наведено граф-схема алгоритму стискання і розархівовування інформації LZW

Проведено тестування утиліти на тестових файлах.

В результаті виконання даної курсової роботи засвоєно стискання інформації методом групового кодування (LZW).

Використана література

* Кричевский Р.Е. Сжатие и поиск информации. – М., Радио и связь, 1989
* Рябко Б.Я. Сжатие информации с помощью стопки книг. // Проблемы передачи информации.- 1980, т.16, №4. С.16-21
* Кохманюк Д. Сжатие данных: как это делается // Компьютеры+программы,1995.
* Хоффман Л. Современные методы защиты информации: Пер. с англ. -М.: Сов. радио, 1980, 264с.
* Зкслер А. Б. Архиваторы (Проґраммы для хранения й обработки информации в сжатом виде). - М: Малое предприятие "Алекс", 1992. - 150 с. 12. Компьютер Пресе. -М: 1994; N 1-5

Додаток (Лістинг програми)

#include <algorithm>

#include <array>

#include <climits>

#include <cstddef>

#include <cstdint>

#include <cstdlib>

#include <exception>

#include <fstream>

#include <ios>

#include <iostream>

#include <istream>

#include <limits>

#include <memory>

#include <ostream>

#include <stdexcept>

#include <string>

#include <utility>

#include <vector>

// Safety macro; if not defined, some overkillish safety checks are avoided.

//#define TAKE\_NO\_RISKS

/// Type used to store and retrieve codes.

using CodeType = std::uint32\_t;

namespace globals {

/// Dictionary Maximum Size (when reached, the dictionary will be reset)

const CodeType dms{ 512 \* 1024 };

} // namespace globals

///

/// @brief Special codes used by the encoder to control the decoder.

/// @todo Metacodes should not be hardcoded to match their index.

///

enum class MetaCode : CodeType {

Eof = 1u << CHAR\_BIT, ///< End-of-file.

};

///

/// @brief Encoder's custom dictionary type.

///

class EncoderDictionary {

///

/// @brief Binary search tree node.

///

struct Node {

///

/// @brief Default constructor.

/// @param c byte that the Node will contain

///

explicit Node(char c) : first(globals::dms), c(c), left(globals::dms), right(globals::dms)

{

}

CodeType first; ///< Code of first child string.

char c; ///< Byte.

CodeType left; ///< Code of child node with byte < `c`.

CodeType right; ///< Code of child node with byte > `c`.

};

public:

///

/// @brief Default constructor.

/// @details It builds the `initials` cheat sheet.

///

EncoderDictionary()

{

const long int minc = std::numeric\_limits<char>::min();

const long int maxc = std::numeric\_limits<char>::max();

CodeType k{ 0 };

for (long int c = minc; c <= maxc; ++c)

initials[static\_cast<unsigned char> (c)] = k++;

vn.reserve(globals::dms);

reset();

}

///

/// @brief Resets dictionary to its initial contents.

/// @note Adds dummy nodes to account for the metacodes.

///

void reset()

{

vn.clear();

const long int minc = std::numeric\_limits<char>::min();

const long int maxc = std::numeric\_limits<char>::max();

for (long int c = minc; c <= maxc; ++c)

vn.push\_back(Node(c));

// add dummy nodes for the metacodes

vn.push\_back(Node('\x00')); // MetaCode::Eof

}

///

/// @brief Searches for a pair (`i`, `c`) and inserts the pair if it wasn't found.

/// @param i code to search for

/// @param c attached byte to search for

/// @return The index of the pair, if it was found.

/// @retval globals::dms if the pair wasn't found

///

CodeType search\_and\_insert(CodeType i, char c)

{

if (i == globals::dms)

return search\_initials(c);

const CodeType vn\_size = vn.size();

CodeType ci{ vn[i].first }; // Current Index

if (ci != globals::dms)

{

while (true)

if (c < vn[ci].c)

{

if (vn[ci].left == globals::dms)

{

vn[ci].left = vn\_size;

break;

}

else

ci = vn[ci].left;

}

else

if (c > vn[ci].c)

{

if (vn[ci].right == globals::dms)

{

vn[ci].right = vn\_size;

break;

}

else

ci = vn[ci].right;

}

else // c == vn[ci].c

return ci;

}

else

vn[i].first = vn\_size;

vn.push\_back(Node(c));

return globals::dms;

}

///

/// @brief Fakes a search for byte `c` in the one-byte area of the dictionary.

/// @param c byte to search for

/// @return The code associated to the searched byte.

///

CodeType search\_initials(char c) const

{

return initials[static\_cast<unsigned char> (c)];

}

///

/// @brief Returns the number of dictionary entries.

///

std::vector<Node>::size\_type size() const

{

return vn.size();

}

private:

/// Vector of nodes on top of which the binary search tree is implemented.

std::vector<Node> vn;

/// Cheat sheet for mapping one-byte strings to their codes.

std::array<CodeType, 1u << CHAR\_BIT> initials;

};

///

/// @brief Helper structure for use in `CodeWriter` and `CodeReader`.

///

struct ByteCache {

///

/// @brief Default constructor.

///

ByteCache() : used(0), data(0x00)

{

}

std::size\_t used; ///< Bits currently in use.

unsigned char data; ///< The bits of the cached byte.

};

///

/// @brief Variable binary width code writer.

///

class CodeWriter {

public:

///

/// @brief Default constructor.

/// @param [out] os Output Stream to write codes to

///

explicit CodeWriter(std::ostream& os) : os(os), bits(CHAR\_BIT + 1)

{

}

///

/// @brief Destructor.

/// @note Writes `MetaCode::Eof` and flushes the last byte to the stream.

///

~CodeWriter()

{

write(static\_cast<CodeType> (MetaCode::Eof));

// write the incomplete leftover byte as-is

if (lo.used != 0)

os.put(static\_cast<char> (lo.data));

}

///

/// @brief Getter for `CodeWriter::bits`.

///

std::size\_t get\_bits() const

{

return bits;

}

///

/// @brief Resets internal binary width.

/// @note Default value is `CHAR\_BIT + 1`.

///

void reset\_bits()

{

bits = CHAR\_BIT + 1;

}

///

/// @brief Increases internal binary width by one.

/// @throws std::overflow\_error internal binary width cannot be increased

/// @remarks The exception should never be thrown, under normal circumstances.

///

void increase\_bits()

{

#ifdef TAKE\_NO\_RISKS

if (bits == SIZE\_MAX)

throw std::overflow\_error("CodeWriter::increase\_bits()");

#endif

++bits;

}

///

/// @brief Writes the code `k` with a binary width of `CodeWriter::bits`.

/// @param k code to be written

/// @return Whether or not the stream can be used for output.

/// @retval true the output stream can still be used

/// @retval false the output stream can no longer be used

///

bool write(CodeType k)

{

std::size\_t remaining\_bits{ bits };

if (lo.used != 0)

{

lo.data |= k << lo.used;

os.put(static\_cast<char> (lo.data));

k >>= CHAR\_BIT - lo.used;

remaining\_bits -= CHAR\_BIT - lo.used;

lo.used = 0;

lo.data = 0x00;

}

while (remaining\_bits != 0)

if (remaining\_bits >= CHAR\_BIT)

{

os.put(static\_cast<char> (k));

k >>= CHAR\_BIT;

remaining\_bits -= CHAR\_BIT;

}

else

{

lo.used = remaining\_bits;

lo.data = k;

break;

}

return true;

}

private:

std::ostream& os; ///< Output Stream.

std::size\_t bits; ///< Binary width of codes.

ByteCache lo; ///< LeftOvers.

};

///

/// @brief Variable binary width code reader.

///

class CodeReader {

public:

///

/// @brief Default constructor.

/// @param [in] is Input Stream to read codes from

///

explicit CodeReader(std::istream& is) : is(is), bits(CHAR\_BIT + 1), feofmc(false)

{

}

///

/// @brief Getter for `CodeReader::bits`.

///

std::size\_t get\_bits() const

{

return bits;

}

///

/// @brief Resets internal binary width.

/// @note Default value is `CHAR\_BIT + 1`.

///

void reset\_bits()

{

bits = CHAR\_BIT + 1;

}

///

/// @brief Increases internal binary width by one.

/// @throws std::overflow\_error if internal binary width cannot be increased

/// @remarks The exception should never be thrown, under normal circumstances.

///

void increase\_bits()

{

#ifdef TAKE\_NO\_RISKS

if (bits == SIZE\_MAX)

throw std::overflow\_error("CodeReader::increase\_bits()");

#endif

++bits;

}

///

/// @brief Reads the code `k` with a binary width of `CodeReader::bits`.

/// @param [out] k code to be read

/// @return Whether or not the stream can be used for input.

/// @retval true the input stream can still be used

/// @retval false the input stream can no longer be used

///

bool read(CodeType& k)

{

// ready-made bit masks

static const std::array<unsigned long int, 9> masks{

{0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF}

};

std::size\_t remaining\_bits{ bits };

std::size\_t offset{ lo.used };

unsigned char temp;

k = lo.data;

remaining\_bits -= lo.used;

lo.used = 0;

lo.data = 0x00;

while (remaining\_bits != 0 && is.get(reinterpret\_cast<char&> (temp)))

if (remaining\_bits >= CHAR\_BIT)

{

k |= static\_cast<CodeType> (temp) << offset;

offset += CHAR\_BIT;

remaining\_bits -= CHAR\_BIT;

}

else

{

k |= static\_cast<CodeType> (temp & masks[remaining\_bits]) << offset;

lo.used = CHAR\_BIT - remaining\_bits;

lo.data = temp >> remaining\_bits;

break;

}

if (k == static\_cast<CodeType> (MetaCode::Eof))

{

feofmc = true;

return false;

}

return true;

}

///

/// @brief Returns if EF is considered corrupted.

/// @retval true didn't find end-of-file metacode

/// @retval false found end-of-file metacode

///

bool corrupted() const

{

return !feofmc;

}

private:

std::istream& is; ///< Input Stream.

std::size\_t bits; ///< Binary width of codes.

bool feofmc; ///< Found End-Of-File MetaCode.

ByteCache lo; ///< LeftOvers.

};

///

/// @brief Computes the minimum number of bits required to store the value of `n`.

/// @param n number to be evaluated

/// @return Number of required bits.

///

std::size\_t required\_bits(unsigned long int n)

{

std::size\_t r{ 1 };

while ((n >>= 1) != 0)

++r;

return r;

}

///

/// @brief Compresses the contents of `is` and writes the result to `os`.

/// @param [in] is input stream

/// @param [out] os output stream

///

void compress(std::istream& is, std::ostream& os)

{

EncoderDictionary ed;

CodeWriter cw(os);

CodeType i{ globals::dms }; // Index

char c;

bool rbwf{ false }; // Reset Bit Width Flag

while (is.get(c))

{

// dictionary's maximum size was reached

if (ed.size() == globals::dms)

{

ed.reset();

rbwf = true;

}

const CodeType temp{ i };

if ((i = ed.search\_and\_insert(temp, c)) == globals::dms)

{

cw.write(temp);

i = ed.search\_initials(c);

if (required\_bits(ed.size() - 1) > cw.get\_bits())

cw.increase\_bits();

}

if (rbwf)

{

cw.reset\_bits();

rbwf = false;

}

}

if (i != globals::dms)

cw.write(i);

}

///

/// @brief Decompresses the contents of `is` and writes the result to `os`.

/// @param [in] is input stream

/// @param [out] os output stream

///

void decompress(std::istream& is, std::ostream& os)

{

std::vector<std::pair<CodeType, char>> dictionary;

// "named" lambda function, used to reset the dictionary to its initial contents

const auto reset\_dictionary = [&dictionary] {

dictionary.clear();

dictionary.reserve(globals::dms);

const long int minc = std::numeric\_limits<char>::min();

const long int maxc = std::numeric\_limits<char>::max();

for (long int c = minc; c <= maxc; ++c)

dictionary.push\_back({ globals::dms, static\_cast<char> (c) });

// add dummy elements for the metacodes

dictionary.push\_back({ 0, '\x00' }); // MetaCode::Eof

};

const auto rebuild\_string = [&dictionary](CodeType k) -> const std::vector<char> \*{

static std::vector<char> s; // String

s.clear();

// the length of a string cannot exceed the dictionary's number of entries

s.reserve(globals::dms);

while (k != globals::dms)

{

s.push\_back(dictionary[k].second);

k = dictionary[k].first;

}

std::reverse(s.begin(), s.end());

return &s;

};

reset\_dictionary();

CodeReader cr(is);

CodeType i{ globals::dms }; // Index

CodeType k; // Key

while (true)

{

// dictionary's maximum size was reached

if (dictionary.size() == globals::dms)

{

reset\_dictionary();

cr.reset\_bits();

}

if (required\_bits(dictionary.size()) > cr.get\_bits())

cr.increase\_bits();

if (!cr.read(k))

break;

if (k > dictionary.size())

throw std::runtime\_error("invalid compressed code");

const std::vector<char>\* s; // String

if (k == dictionary.size())

{

dictionary.push\_back({ i, rebuild\_string(i)->front() });

s = rebuild\_string(k);

}

else

{

s = rebuild\_string(k);

if (i != globals::dms)

dictionary.push\_back({ i, s->front() });

}

os.write(&s->front(), s->size());

i = k;

}

if (cr.corrupted())

throw std::runtime\_error("corrupted compressed file");

}

///

/// @brief Prints usage information and a custom error message.

/// @param s custom error message to be printed

/// @param su Show Usage information

///

void print\_usage(const std::string& s = "", bool su = true)

{

if (!s.empty())

std::cerr << "\nERROR: " << s << '\n';

if (su)

{

std::cerr << "\nUsage:\n";

std::cerr << "\tprogram -flag input\_file output\_file\n\n";

std::cerr << "Where `flag' is either `c' for compressing, or `d' for decompressing, and\n";

std::cerr << "`input\_file' and `output\_file' are distinct files.\n\n";

std::cerr << "Examples:\n";

std::cerr << "\tCourseWorkLZW.exe -c license.txt license.lzw\n";

std::cerr << "\tCourseWorkLZW.exe -d license.lzw new\_license.txt\n";

}

std::cerr << std::endl;

}

///

/// @brief Actual program entry point.

/// @param argc number of command line arguments

/// @param [in] argv array of command line arguments

/// @retval EXIT\_FAILURE for failed operation

/// @retval EXIT\_SUCCESS for successful operation

///

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

{

if (argc != 4)

{

print\_usage("Wrong number of arguments.");

return EXIT\_FAILURE;

}

enum class Mode {

Compress,

Decompress

};

Mode m;

if (std::string(argv[1]) == "-c")

m = Mode::Compress;

else

if (std::string(argv[1]) == "-d")

m = Mode::Decompress;

else

{

print\_usage(std::string("flag `") + argv[1] + "' is not recognized.");

return EXIT\_FAILURE;

}

const std::size\_t buffer\_size{ 1024 \* 1024 };

// these custom buffers should be larger than the default ones

const std::unique\_ptr<char[]> input\_buffer(new char[buffer\_size]);

const std::unique\_ptr<char[]> output\_buffer(new char[buffer\_size]);

std::ifstream input\_file;

std::ofstream output\_file;

input\_file.rdbuf()->pubsetbuf(input\_buffer.get(), buffer\_size);

input\_file.open(argv[2], std::ios\_base::binary);

if (!input\_file.is\_open())

{

print\_usage(std::string("input\_file `") + argv[2] + "' could not be opened.");

return EXIT\_FAILURE;

}

output\_file.rdbuf()->pubsetbuf(output\_buffer.get(), buffer\_size);

output\_file.open(argv[3], std::ios\_base::binary);

if (!output\_file.is\_open())

{

print\_usage(std::string("output\_file `") + argv[3] + "' could not be opened.");

return EXIT\_FAILURE;

}

try

{

input\_file.exceptions(std::ios\_base::badbit);

output\_file.exceptions(std::ios\_base::badbit | std::ios\_base::failbit);

if (m == Mode::Compress)

compress(input\_file, output\_file);

else

if (m == Mode::Decompress)

decompress(input\_file, output\_file);

}

catch (const std::ios\_base::failure& f)

{

print\_usage(std::string("File input/output failure: ") + f.what() + '.', false);

return EXIT\_FAILURE;

}

catch (const std::exception& e)

{

print\_usage(std::string("Caught exception: ") + e.what() + '.', false);

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}