ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи

на ступінь вищої освіти бакалавр

на тему: **«РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АРХІВАЦІЇ ДАНИХ НА ОСНОВІ АЛГОРИТМУ СІМЕЙСТВА LZ77 ТА ЕНТРОПІЙНОГО КОДУВАЛЬНИКА МОВОЮ C++»**

Виконав студент 5 курсу, групи ППЗ-51

спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Сіраченко Д.В.
(прізвище та ініціали)

Керівник Аверічев І.М.
(прізвище та ініціали)

Рецензент _____
(прізвище та ініціали)

Київ-2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення_____
Ступінь вищої освіти - «Бакалавр»_____
Спеціальність - 121 «Інженерія програмного забезпечення»_____

ЗАТВЕРДЖУЮ
Доцент кафедри
Інженерії програмного забезпечення
_____І.М. Аверічев_
«____» _____2023 року

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

_____**СІРАЧЕНКО ДМИТРІЯ ВЛАДИСЛАВОВИЧА**_____
(прізвище, ім'я, по батькові)

1. Тема роботи: «Розробка програмного забезпечення для архівації даних на основі алгоритму сімейства LZ77 та ентропійного кодувальника мовою С++»_____
Керівник роботи: Аверічев І.М., к.е.н., доцент_____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від «24» лютого 2023 року № 26.

2. Строк подання студентом роботи «1» червня 2023 року.

3. Вхідні дані до роботи:
3.1. Середовище розробки
3.2. Алгоритм дії програми-архіватора

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
4.1. Аналіз та порівняння існуючих прототипів
4.2. Дослідження програмних засобів для розробки програми-архіватора
4.3. Програмна реалізація програми-архіватора
4.4. Приклади використання та тестування системи
4.5. Висновки

5. Перелік демонстраційного матеріалу:

5.1. Титульний слайд
5.2. Мета, об’єкт та предмет дослідження
5.3. Аналіз існуючих рішень
5.4. Вимоги до програми-архіватора
5.5. Програмні засоби реалізації
5.6. Діаграми
5.7. Екранні форми
5.8. Висновки

6. Дата видачі завдання: «25» лютого 2023 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	25.02.2023 - 01.03.2023	Виконано
2	Вивчення існуючих засобів для стиснення даних	01.03.2023 - 15.03.2023	Виконано
3	Проектування архітектури програми-архіватору	15.03.2023 - 15.04.2023	Виконано
4	Реалізація програми-архіватору	15.04.2023 - 10.05.2023	Виконано
5	Оформлення роботи	10.05.2023 - 25.05.2023	Виконано
6	Попередній захист роботи	25.05.2023 - 26.05.2023	Виконано
7	Здача роботи	01.06.2023	Виконано

Студент _____ Сіраченко Д.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Аверічев І.М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи

Ключеві слова: програма-архіватор, LZ77, метод Хаффмена, C++, Visual Studio Code, MinGW, Cmake.

Об’єкт дослідження: процес архівування файлів файлової системи.

Предмет дослідження: додаток, що використовує швидкісний алгоритм стиснення даних LZ4 з ентропійним кодуванням для архівування файлів файлової системи.

Мета роботи: підвищення швидкості архівування файлів файлової системи за рахунок алгоритму LZ4 та якості стиснення за рахунок ентропійного кодування.

Для реалізації поставленної мети потрібно вирішити наступні завдання:

1. Проаналізувати існуючі засоби для архівування файлів файлової системи та основні потреби користувачів програм-архіваторів
2. Розробити вимоги до програми-архіватору
3. Спроектувати програму-архіватор
4. Створити програму-архіватор мовою C++ на основі аналізу основних потреб користувачів
5. Протестувати програму-архіватор

Практичне значення отриманих результатів полягає у написанні основного функціоналу програми-архіватора мовою програмування C++.

В роботі розглянуто основні етапи створення програми-архіватора і досліджено можливості технічних засобів для розробки програми-архіватора.

Розроблено додаток, що втілює основні вимоги користувача, за допомогою алгоритму LZ4 та ентропійного кодування.

Галузь використання — розроблений додаток може бути використаний користувачами комп’ютерних систем для стиснення будь яких файлів файлової системи.

ЗМІСТ

Вступ

Програми-архіватори, або архіватори — дуже корисні інструменти для роботи з файлами на комп'ютері, особливо якщо потрібно зменшити обсяг дискового простору або передати файли через Інтернет. Ці програми працюють за допомогою алгоритмів стиснення даних, що дозволяє зменшити розмір файлів без втрати якості.

Одними з найпопулярніших програм-архіваторів, які використовуються сьогодні, є WinRAR, 7-Zip і WinZip. Ці програми можуть працювати з багатьма форматами файлів, включаючи ZIP, RAR, TAR, GZIP і CAB.

Однією з головних переваг використання архіваторів є можливість стиснення файлів. Архіватори використовують різні методи стиснення, які дозволяють зменшити розмір файлу без втрати якості даних. Це особливо корисно, якщо є багато файлів, які потрібно зберігати на обмеженому дисковому просторі. Стиснуті файли займає менше місця на диску, що знижує навантаження на жорсткий диск і підвищує швидкість роботи комп'ютера. Крім того, архіватори дозволяють захищати файли паролями і шифруванням. Це допомагає захистити файли від несанкціонованого доступу. Захищені архіви можуть бути корисними для зберігання конфіденційної інформації, такої як паролі, банківські реквізити, документи з особистою інформацією тощо.

Архіватори файлів також можуть бути корисними для групування файлів в одну папку архіву, що дозволяє легко керувати багатьма файлами. Це особливо корисно, якщо є багато пов’язаних один з одним файлів.

Ще однією перевагою використання програм-архіваторів є зручність передачі файлів через Інтернет. Архіватори дозволяють легко створювати архіви з великою кількістю файлів, що зменшує кількість файлів, які необхідно перенести, і збільшує швидкість передачі файлів.

Швидкість роботи архіваторів залежить від розміру файлів, які потрібно стиснути, а також від обраного алгоритму стиснення. Звичайно, архіватори

можуть швидко стискати файли, особливо якщо використовувати їх з оптимізованими налаштуваннями. однак, якщо стискаються великі файли, це може зайняти трохи більше часу.

Швидкість роботи архіватора файлів важлива з кількох причин. По-перше, чим швидше архіватор зможе стискати файли, тим менше витрачається часу на ці файли, що особливо важливо для збереження, якщо ведеться робота з великими обсягами даних. По-друге, якщо використовується архіватор для передачі файлів через Інтернет або іншим способом, швидкість архіватора може бути використана для часу, необхідного для завантаження та розпакування файлів.

Відповідно, швидкість роботи архіватора є важливим фактором, який слід враховувати при виборі програми для стиснення та розпакування файлів.

Щоб розробити власну програму-архіватор, можна використовувати інструменти C++, MinGW і CMake. C++ - це об'єктно-орієнтована мова програмування, яка дозволяє розробляти програми, що працюють з різними типами даних [2]. MinGW — це набір інструментів, який дозволяє скомпілювати програму з коду C++ у Windows. CMake — засіб, що дозволяє автоматизувати процес компіляції програм.

1 АНАЛІЗ ЗАСОБІВ ДЛЯ ВИРІШЕННЯ ПРОБЛЕМИ АРХІВУВАННЯ ФАЙЛІВ

1.1 Цифрова інформація

Термін "інформація" походить від латинського слова "informatio", що означає відомості, роз'яснення, виклад. Незважаючи на широке поширення цього терміна, поняття інформації є одним із самих дискусійних у науці. У цей час наука намагається знайти загальні властивості й закономірності, які відповідають багатогранному поняттю інформація, але поки це поняття багато в чому залишається інтуїтивним й одержує різні значеннєві наповнення в різних галузях людської діяльності [5].

Цифрова інформація — це тип інформації, що представлений серією двійкових цифр або бітів. Двома можливими значеннями біта є 0 і 1, які можна використовувати для представлення різних типів інформації, наприклад тексту, зображень і звуку. Цифрова інформація присутня усюди в сучасному суспільстві та використовується в великій кількості додатків, від спілкування та розваг до комерції та науки.

Цифрова інформація представлена серією бітів, які можуть бути організовані в групи по 8 бітів або байтів. Кожен байт може представляти один символ тексту, піксель у зображенні або зразок звуку. Комбінуючи байти, можна представити більш складні типи інформації, такі як рядки тексту, зображення та аудіофайли.

Однією з ключових переваг цифрової інформації є те, що її можна легко копіювати та передавати. Цифрову інформацію можна передавати через такі мережу Інтернет, і зберігати та отримувати за допомогою цифрових пристроїв зберігання, таких як жорсткі диски та флеш-накопичувачі.

Цифрова інформація використовується в великій кількості програм, включаючи комунікацію, розваги, комерцію та науку. У комунікації цифрова інформація використовується для передачі повідомлень між комп'ютерами та пристроями, а також дозволяє використовувати електронну пошту, обмін миттєвими повідомленнями та відеоконференції. У сфері розваг цифрова інформація використовується для створення та розповсюдження фільмів, музики та відеоігор. У торгівлі цифрова інформація використовується для полегшення транзакцій, таких як онлайн-магазини та банківські операції. У науці цифрова інформація використовується для збору та аналізу даних і використовується в таких галузях, як генетика, астрономія та метеорологія.

Цифрова інформація — це тип інформації, що представлена за допомогою двійкових чисел. Цифрова інформація досить широко використовується у сучасному суспільстві та в великій кількості додатків, включаючи зв'язок, розваги, комерцію та науку.

1.2 Кількість інформації

Кількість інформації дає відповідь на те, якою повинна бути ємність накопичувача для зберігання даних. Кількісна оцінка інформації пов'язана з поняттям ентропії [9].

Інформаційні дані вимірюються у одиницях: біт (1 біт), байт (8 біт), кібібайт (KiB = 1024 байт), мебібайт (MiB = 1024 кібібайт), гібібайт (GiB = 1024 мебібайт) і тд.

1.3 Інформаційна ентропія

Ентропія є мірою невизначеності, непрогнозованості ситуації [9]. Простими словами, це повідомляє нам, скільки інформації потрібно для опису даних.

Зменшення ентропії, що відбулось завдяки деякому повідомленню, точно збігається з кількістю інформації, яка міститься в цьому повідомленні [9].

За Шенноном (Клод Шеннон) інформаційну ентропію визначають як абсолютну межу найкращого стиснення даних без втрат.

Інформаційну ентропію за Шенноном визначають:

$$H(X) = -\sum_{n=1}^{N} p_n \log_2 p_n \qquad (1.3.1)$$

де H(X) — ентропія набору даних X, p(x) — імовірність кожного можливого результату x у X, а log2 — логарифм за основою 2.

Тому одиницю виміру інформації називають біт або Шеннон, Shannon, bit (binary information).

Загалом ентропія Шеннона є корисним інструментом для кількісного визначення кількості інформації в наборі даних і може використовуватися в різноманітних програмах, таких як стиснення даних, криптографія та теорія інформації.

1.4 Системи числення

Система числення — сукупність способів і засобів запису чисел для проведення підрахунків [10]. Серед систем числення є позиційні — система числення, у якій значення цифри залежить від її позиції в записі числа.

Представлене позиційною системою число ділиться на розряди — позицію цифри у записі числа. Також система числення містить основу — кількість цифр в алфавіті. Для десяткової основа дорівнює 10, для шістнадцяткової 16, для двійкової основа дорівнює 2.

Загальноприйнятою для людей є десяткова система числення, однак сучасні обчислювальні пристрої використовують двійкову систему числення. Також

програмістами широко використовується шістнадцяткова система числення, оскільки за допомогою неї можливо компактно записати значення ступенів двійки.

Таблиця 1.4.1 — Представлення цифр у різних системах числення

Десяткова система числення	Шістнадцяткова система числення	Двійкова система числення
1	1	1
10	0xA	0b1010
255	0xFF	0b11111111

1.4.1 Двійкова система числення

Двійкова система числення є основою сучасних цифрових обчислень і широко використовується в усіх аспектах апаратного та програмного забезпечення комп’ютерів.

Двійкові цифри (біти) використовуються для представлення цифрових даних у пам’яті та сховищі комп’ютера. Кожен біт може бути в одному з двох станів, 0 або 1, які можна використовувати для представлення двійкових даних, таких як числа, літери та символи.

Двійкова система використовується для представлення комп’ютерних інструкцій і даних у формі машинного коду, який виконується процесором для виконання таких операцій, як арифметичні, логічні та введення/виведення.

Двійкова система використовується в цифрових комунікаціях, таких як передача даних через Інтернет і комп’ютерні мережі. Дані передаються у вигляді двійкових сигналів, які інтерпретуються приймальним пристроєм для відновлення вихідних даних.

Отже, двійкова система числення є важливим компонентом сучасних цифрових обчислень і використовується в усіх аспектах комп’ютерного апаратного та програмного забезпечення.

1.5 Кодування інформації

Кодування полягає у перетворенні інформації на впорядкований набір символів. При кодуванні кожному повідомленню ставиться відповідна кодова комбінація − набір символів. Множина повідомлень називається алфавітом повідомлень, а множина символів для кодування називається алфавітом джерела, або вторинним алфавітом [11].

Таблиця 1.5.1 — Коди фіксованої довжини для випадкових символів

Алфавіт повідомлень [11]	Двійкова система [11]	Код Фібоначчі	Префіксний код
A	000	000	0
B	001	001	010
C	010	011	011

В даному випадку наведено коди фіксованої довжини з двійковою системою числення, яка використовується сучасними обчислювальними пристроями, але вибір системи числення не відіграє ключової ролі, оскільки для фіксованого коду важливо максимальне використання усіх можливих значень дійсних кодових слів та однозначне декодування.

Під час кодування використовується одна із систем числення, на сьогоднішній день, сучасні комп’ютери працюють з двійковою системою числення, тому будь-які дані представляються у вигляді нулей та одиниць (0 та 1).

1.5.1 Префіксний код

Префіксний код — це будь-яка схема кодування змінної довжини (схема кодування, яка використовує кодові слова різної довжини для кодування різних символів), з важливою властивістю того що, жодне кодове слово не є префіксом

іншого. Префіксом рядка є підрядок цього рядка, який знаходиться на початку рядка. Ця властивість гарантує, що код може бути декодований однозначно, без необхідності будь-яких спеціальних роздільників або роздільників між кодовими словами.

Таблиця 1.5.1.1 — Префіксні коди для випадкових символів

Символ	Префіксний код
A	0
B	10
C	11

Префіксні коди широко використовуються в алгоритмах стиснення даних, зокрема в алгоритмах стиснення без втрат, таких як кодування Хаффмена.

Властивість префікса коду гарантує, що жодні два кодових слова не мають спільного префікса, що означає, що код може бути однозначно декодований.

Префіксні коди є потужним інструментом для стиснення даних, що забезпечує ефективне кодування символів на основі їх частоти появи у вхідних даних (кодування Хаффмена). Використання префіксних кодів, особливо в поєднанні з іншими методами стиснення, зробило революцію в галузі стиснення даних і дозволило зберігати та ефективно передавати великі обсяги даних.

1.6 Структури даних

Серед структур даних можна навести наступні: **лінійний масив**, **двійкове дерево** та **хеш-таблицю**.

Лінійний масив — це структура даних, яка зберігає набір елементів одного типу в безперервному блоці пам’яті. Це тип масиву, який розміщено в одному рядку, і до кожного елемента в масиві доступ доступний за його індексом.

Лінійний масив визначається з фіксованим розміром, який є кількістю елементів, які він може містити. Перший елемент у масиві зберігається під

індексом “0”, другий елемент під індексом “1” і так далі. Кожен елемент у масиві зберігається в послідовних місцях пам’яті, а адреса пам’яті першого елемента в масиві використовується як посилання для доступу до будь-якого елемента в масиві.

У стисненні даних лінійні масиви часто використовуються в алгоритмах стиснення без втрат, таких як кодування Хаффмена, яке є популярним методом стиснення текстових файлів.

Лінійні масиви також можна використовувати для лінійного зберігання та обробки даних, тобто елементи масиву зберігаються в пам’яті послідовно. Це полегшує доступ і ефективну обробку даних. Наприклад, лінійні масиви можна використовувати для обробки сигналів у програмах обробки сигналів, де дані зазвичай зберігаються та обробляються лінійним способом.

Загалом, лінійні масиви є простим і ефективним способом організації та обробки даних лінійним способом, що робить їх популярною структурою даних у різноманітних програмах.

Двійкове дерево — це деревовидна структура даних, у якій кожен вузол має не більше двох дочірніх елементів, які називаються лівим і правим дочірніми елементами. Двійкове дерево можна представити за допомогою лінійного масиву за допомогою техніки, яка називається обходом рівня в порядку або обходом у ширину.

При проходженні в порядку рівня вузли двійкового дерева зберігаються в масиві рівень за рівнем, зліва направо. Кореневий вузол зберігається за першим індексом масиву, а лівий і правий дочірні елементи кожного вузла зберігаються за двома наступними індексами масиву. Якщо вузол не має лівого чи правого дочірнього елемента, відповідний індекс у масиві встановлюється на спеціальне значення, щоб вказати, що дочірній елемент не існує.

Щоб вставити новий вузол у двійкове дерево, що зберігається в масиві, спочатку виконується пошук у масиві, щоб знайти першу доступну позицію в кінці обходу порядку рівня. Новий вузол вставляється в цю позицію, а потім масив

оновлюється, щоб зберегти властивість двійкового дерева.

Для обходу двійкового дерева, представленого у вигляді масиву, можна використовувати обхід рівня. У цьому методі вузли двійкового дерева відвідуються в порядку рівня, зліва направо, шляхом ітерації по масиву.

Хоча цей метод представлення двійкового дерева є простим і легким для розуміння, він вимагає більше пам’яті, ніж інші методи, оскільки він використовує однакову кількість простору незалежно від кількості вузлів у дереві. Тому цей метод зазвичай не використовується на практиці для великих двійкових дерев.

Складність вставки, видалення та пошуку для двійкового дерева:

Таблиця 1.6.2.1 — Складність алгоритму для двійкового дерева

	Вставка	Видалення	Пошук
Складність алгоритму (average case)	O(log n)	O(log n)	O(log n)

В інформатиці двійкові дерева широко використовуються як структура даних для зберігання та організації даних.

У структурах даних двійкові дерева зазвичай використовуються для реалізації ряду інших структур даних, таких як бінарні дерева пошуку, дерева купи та дерева AVL. Двійкові дерева пошуку використовуються для ефективного пошуку даних у відсортованому списку, тоді як дерева купи використовуються для ефективного збереження колекції даних у певному порядку.

У стисненні даних двійкові дерева використовуються як в алгоритмах стиснення без втрат, так і з втратами. Наприклад, у стисненні без втрат алгоритм кодування Хаффмена використовує двійкові дерева для ефективного стиснення даних. У стисненні з втратами алгоритм стиснення JPEG використовує двійкові дерева для виконання частотного аналізу даних зображення та ефективного їх стиснення.

Хеш-таблиця — це структура даних, яка використовується для швидкого й ефективного зберігання й отримання даних. Він використовує техніку, що називається хешуванням, щоб зіставляти ключі з відповідними значеннями.

Хешування — це процес перетворення значення ключа в індекс масиву, який називається хеш-таблицею, який можна використовувати для пошуку пов’язаного значення.

У хеш-таблиці з використанням лінійних масивів хеш-таблиця реалізована у вигляді масиву фіксованого розміру. Кожен елемент хеш-таблиці — це “бакет”, яке може містити одну або кілька пар ключ-значення. Ключі хешуються до індексу масиву, а відповідне значення зберігається у “бакеті” за цим індексом. Якщо два ключі хешують один і той самий індекс, що називається “зіткненням”, значення зберігаються в одному сегменті за допомогою зв’язаного списку або іншої структури даних.

Щоб вставити пару ключ-значення в хеш-таблицю, ключ хешується до індексу масиву, а відповідне значення зберігається в сегменті за цим індексом. Щоб отримати значення, пов’язане з заданим ключем, ключ знову хешується, щоб знайти індекс відповідного сегмента, а значення дізнається з сегмента.

Складність вставки, видалення та пошуку для хеш-таблиці (average case):

Таблиця 1.6.3.1 — Складність алгоритму для хеш-таблиці

	Вставка	Видалення	Пошук
Складність алгоритму (average case)	O(1)	O(1)	O(1)

Хеш-таблиці використовуються в широкому діапазоні програм, де важливі ефективний пошук і отримання даних.

В інформатиці хеш-таблиці використовуються для реалізації різних структур даних, таких як словники, асоціативні масиви та таблиці символів. Ці структури

даних широко використовуються в мовах програмування та програмних структурах для ефективного зберігання та отримання даних.

Хеш-таблиці можна використовувати для стиснення даних для підвищення ефективності алгоритму стиснення. Одним із прикладів цього є алгоритм LZ77, який є популярним алгоритмом стиснення даних без втрат, який використовує ковзне вікно та хеш-таблицю для стиснення даних.

Хеш-таблиці є ключовим компонентом багатьох алгоритмів стиснення даних, що забезпечує ефективне кодування повторюваних шаблонів і покращує загальний коефіцієнт стиснення алгоритму.

1.7 Стиснення даних

Стиснення даних дозволяє зменшити розмір даних, щоб зробити їх більш ефективними для зберігання або передачі. Стиснення даних є популярним, оскільки допомагає заощадити місце для зберігання, скоротити час передачі та підвищити загальну продуктивність роботи програм.

Лінійні масиви та бінарні дерева є двома поширеними структурами даних, які використовуються в алгоритмах стиснення даних. Лінійні масиви, які по суті є одновимірними масивами, часто використовуються в алгоритмах стиснення без втрат, таких як кодування Хаффмена. З іншого боку, двійкові дерева використовуються як в алгоритмах без втрат, так і в алгоритмах із втратами, наприклад, стиснення JPEG.

Алгоритми без втрат (lossless algorithms) гарантують, що оригінальні дані можна ідеально реконструювати зі стиснених даних. Приклади алгоритмів без втрат включають кодування Хаффмена та стиснення Лемпеля-Зіва (LZ77). Алгоритми з втратами (lossy algorithms), з іншого боку, допускають певну втрату даних під час стиснення, щоб досягти більшого коефіцієнта стиснення. Приклади алгоритмів із втратами включають стиснення JPEG і MP3.

Основна відмінність між алгоритмами без втрат і алгоритмами з втратами полягає в тому, що алгоритми без втрат зберігають усі вихідні дані, тоді як алгоритми з втратами жертвують деякими вихідними даними, щоб досягти більшого коефіцієнта стиснення.

Основними властивостями алгоритмів стиснення даних є коефіцієнт стиснення, швидкість кодування та декодування, обсяг пам’яті необхідний для роботи алгоритму.

Проблема багатьох алгоритмів стиснення полягає в тому, що вони досить вимогливі до ресурсів процесора (обчислювальних та пам’яті), однак існують альтернативи, що дозволяють стискати дані з мінімальними навантаженнями на обчислювальний пристрій.

На жаль, не існує жодного чудового алгоритму стиснення на всі види даних. Натомість існує кілька різних алгоритмів, що спеціалізовані на певні типи даних. Наприклад, кодування Хаффмена та алгоритми сімейства LZ77 дозволяють зменшити розмір тексту майже на 50%, тоді коли, наприклад, алгоритми MPEG націлені на стискання даних мультимедія (зображення, аудіо тощо).

В області стиснення даних діє закон важеля, тобто алгоритми, що використовують більше ресурсів, досягають кращої якості стиснення, а менш ресурсомісткі алгоритми, навпаки, за якістю стиснення, як правило, поступаються більш ресурсомістким [12].

1.7.1 Алгоритм LZ77

Алгоритм LZ77 — це популярний алгоритм стиснення даних без втрат, який працює шляхом пошуку повторюваних шаблонів у вхідних даних і кодування їх як посилань на дані, що зустрічалися раніше.

Алгоритм починається з порожнього ковзного вікна та буфера перегляду, який містить кілька перших символів вхідних даних.

Алгоритм шукає найдовший збіг між символами в буфері попереднього

перегляду та символами в ковзному вікні.

Коли збіг знайдено, алгоритм виводить пару (відстань, довжину), яка описує відповідний підрядок. Відстань — це кількість символів між початком відповідного підрядка та поточною позицією у вхідних даних. Довжина — це довжина зіставленого підрядка.

Ковзне вікно пересувається вперед на довжину відповідного підрядка, а буфер попереднього перегляду заповнюється наступним набором символів із вхідних даних.

Якщо відповідності не знайдено, алгоритм виводить пару (0, наступний символ), де другим елементом пари є наступний символ у вхідних даних.

Алгоритм продовжує повторювати кроки 2-5, доки всі вхідні дані не будуть стиснуті.

Термінологія, яка використовується в алгоритмі LZ77:

“Ковзне вікно”: вікно фіксованого розміру, яке містить частину попередньо стиснутих даних. Вікно рухається вперед у міру виконання алгоритму.

“Буфер” попереднього перегляду: буфер, який містить кілька наступних символів вхідних даних, які ще не були стиснуті.

“Відстань”: кількість символів між початком відповідного підрядка та поточною позицією у вхідних даних.

“Довжина”: довжина відповідного підрядка.

“Збіг”: пара (відстань, довжина), яка описує відповідний підрядок у вхідних даних.

“Вихід”: стиснуті дані, створені алгоритмом, які складаються з послідовності збігів та окремих символів.

1.7.2 Метод Хаффмена

Алгоритм Хаффмена — це алгоритм стиснення даних без втрат, який працює шляхом кодування кожного символу у вхідних даних за допомогою

кодового слова змінної довжини на основі частоти його появи в даних.

Алгоритм починається з підрахунку частоти появи кожного символу у вхідних даних.

Потім алгоритм будує двійкове дерево, яке називається деревом Хаффмена, яке має символи як листя, а кожен батьківський вузол представляє комбінацію двох дочірніх вузлів із найменшою частотою.

Алгоритм призначає кодові слова кожному символу на основі його позиції в дереві Хаффмена. Символам, розташованим ближче до кореня дерева, призначаються коротші кодові слова, тоді як символам, розташованим далі від кореня, призначаються довші кодові слова.

Стиснуті дані створюються шляхом заміни кожного символу у вхідних даних відповідним кодовим словом.

Термінологія, яка використовується в алгоритмі Хаффмена:

Символ: символ або група символів у вхідних даних, які потрібно закодувати.

Частота: кількість разів, коли символ зустрічається у вхідних даних.

Кодове слово: код змінної довжини, призначений символу на основі його позиції в дереві Хаффмена.

Дерево Хаффмена: двійкове дерево, в якому кожен листовий вузол представляє символ, а кожен внутрішній вузол представляє комбінацію двох дочірніх вузлів із найменшою частотою.

1.8 Аналіз існуючих програм-архіваторів

Одним із рішень для роботи з цифровою інформацією є використання програм-архіваторів або, просто, архіваторів. Архіватор — це програма, призначена для стиснення файлів до меншого розміру, що полегшує зберігання

та передачу даних. Архіватори працюють з файлами. Файлом є одиниця інформації, яка зберігається на комп’ютері, як правило, на вторинному носії інформації, такому як жорсткий диск або флеш-накопичувач. Файл може містити будь-який тип даних, включаючи текст, зображення, аудіо, відео та програмний код. Архіватори аналізують дані у файлі та знаходять шаблони, які можна стиснути. Видаляючи зайві або непотрібні дані, архіватори можуть зменшити розмір файлів на 90% і більше. Це зменшує обсяг пам’яті, необхідного для зберігання файлів, і полегшує передачу великих файлів через мережу Інтернет.

Архіватори особливо корисні для керування цифровою інформацією в умовах обмеженого простору для зберігання. Наприклад, якщо потрібно зберігати велику кількість файлів на жорсткому диску або флеш-пам’яті, використання архіватора може допомогти заощадити місце та зберегти всі файли впорядкованими. Крім того, архіватори файлів можна використовувати для зберігання файлів у стисненому форматі, що може зменшити обсяг місця, необхідного для зберігання резервних копій даних. Це може бути особливо корисним, якщо потрібно регулярно створювати резервні копії великих обсягів даних, наприклад, для бізнесу чи дослідницького проекту.

Ще однією перевагою архіваторів є їх здатність стискати кілька файлів в один архівний файл. Це полегшує керування та передачу кількох файлів одночасно. Наприклад, якщо потрібно надіслати велику кількість файлів, можна скористатися архіватором файлів, щоб об’єднати всі файли в один стиснутий архівний файл. Це зменшує час, необхідний для передачі файлів, і полегшує керування файлами одержувачу.

Таким чином, файлові архіватори є корисним інструментом для керування цифровою інформацією, оскільки вони можуть зменшити розмір файлів і полегшити зберігання та передачу великих обсягів даних. Використовуючи архіватори файлів, можна заощадити місце на жорсткому диску або флеш-накопичувачі, упорядковувати файли та швидко й ефективно передавати великі обсяги даних.

Процес архівування файлів означає стиснення та пакування одного або кількох файлів в один менший файл (архів), що полегшує зберігання та передачу. Архівування також можна використовувати для впорядкування файлів для цілей копіювання шляхом створення одного файлу архіву, який містить кілька файлів.

Існує велика кількість архіваторів, як безкоштовних, так і платних. Одними з найпопулярніших є WinZip, 7-Zip, WinRAR і PeaZip. Ці архіватори популярні, оскільки пропонують широкий спектр функцій, включаючи можливість стискати файли для економії місця на жорсткому диску або флеш-накопичувачі, створювати зашифровані архіви для додаткової безпеки та розпаковувати файли із стислих архівів. Вони також мають зручний інтерфейс та прості в навігації. Крім того, ці програми сумісні з більшістю операційних систем, що робить їх популярними серед користувачів.

1.8.1 WinZip

WinZip — це популярне програмне забезпечення яке існує з 1991 року. Воно широко використовується в операційних системах Windows і Mac. Має зручний інтерфейс, підтримує шифрування для захисту файлів, інтеграцію з хмарними службами зберігання (такі як, Dropbox, Google Drive та OneDrive). WinZip підтримує велику кількість форматів.

Алгоритм WinZip DEFLATE — це метод стиснення даних без втрат, який використовується для зменшення розміру файлів без втрати даних. DEFLATE працює, аналізуючи вхідні дані, щоб знайти повторювані шаблони та замінити їх коротшими символами. Цей метод особливо ефективний для текстових файлів, де часто багато повторюваних рядків символів.

WinZip використовує модифіковану версію алгоритму DEFLATE під назвою “покращений DEFLATE”, який забезпечує швидше стиснення та декомпресію, ніж стандартний алгоритм DEFLATE. Ця модифікація використовує алгоритм LZ77 і швидшу техніку кодування Хаффмена для досягнення більшої продуктивності.

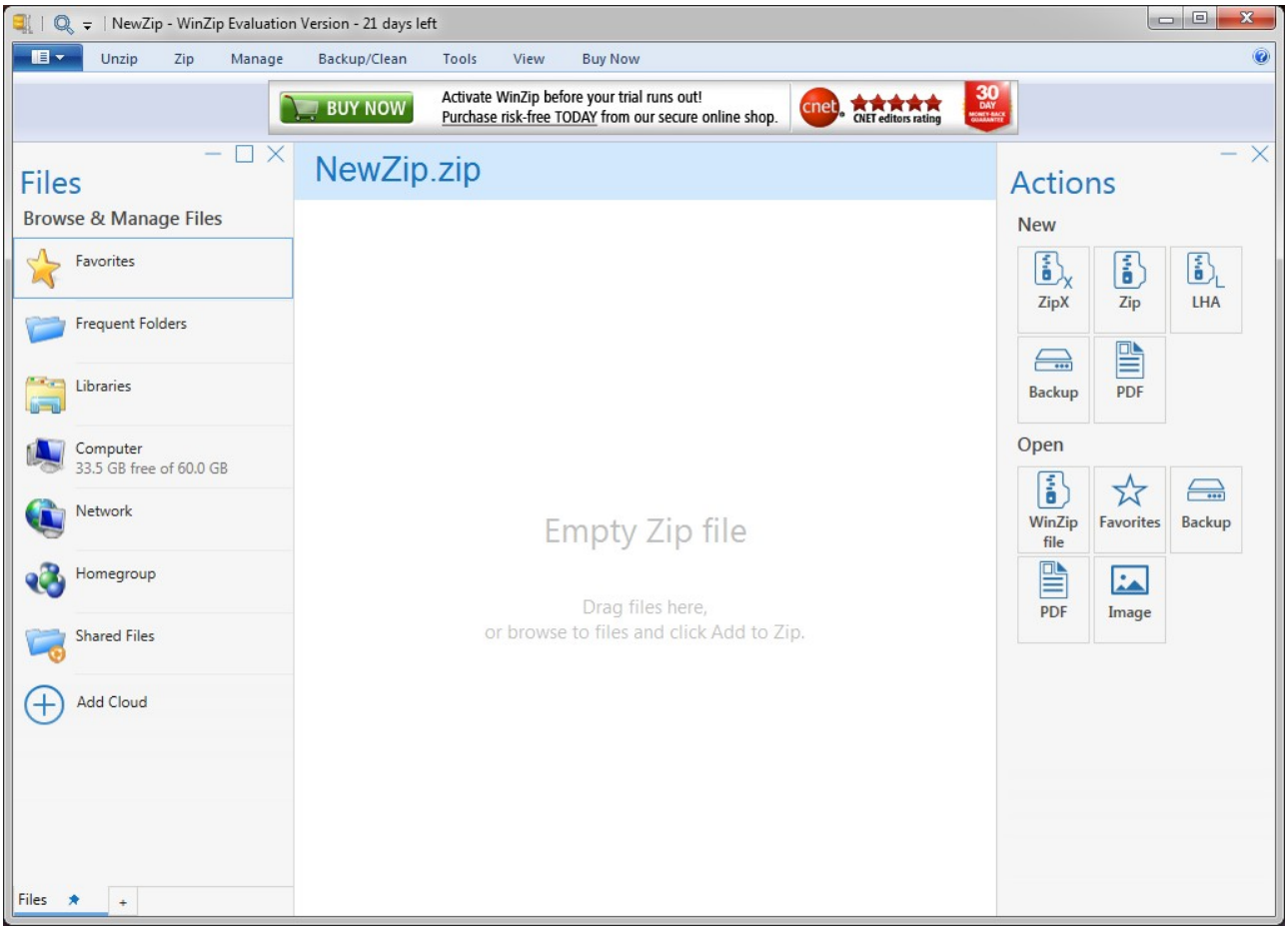


Рисунок 1.8.1.1 — Інтерфейс безкоштовної версії програми WinZip

1.8.2 7-Zip

7-Zip — це безкоштовний архіватор файлів із відкритим кодом.. Він був розроблений Ігорем Павловим у 1999 році та доступний для операційних систем Windows, Linux і macOS.

7-Zip використовує власний формат архіву 7Z, який розроблений для забезпечення більш вищого коефіцієнта стиснення, ніж інші формати архівів, такі як ZIP і RAR. Він також підтримує інші формати файлів, зокрема ZIP, RAR, TAR та ISO, що робить його універсальним інструментом для керування файлами. Також, однією з чудових особливостей 7-Zip є його здатність інтегруватися з оболонкою Windows, що дозволяє легко архівувати та розпаковувати файли з інтерфейсу Windows Explorer.

7-Zip використовує власний алгоритм LZMA який працює, спочатку розділяючи вхідні дані на невеликі блоки, а потім стискаючи кожен блок окремо за допомогою комбінації двох різних методів: алгоритму Лемпеля-Зіва та алгоритму Markov Chain. Алгоритм Лемпеля-Зіва використовується для пошуку повторюваних шаблонів у даних, тоді як алгоритм Markov Chain використовується для прогнозування ймовірності кожного символу в даних на основі його контексту.

Алгоритм LZMA відомий своїм високим коефіцієнтом стиснення, що означає, що він може стискати файли до меншого розміру, ніж інші алгоритми стиснення, зберігаючи при цьому якість вихідних даних. Однак це відбувається за рахунок меншої швидкості стиснення та розпакування порівняно з іншими алгоритмами.

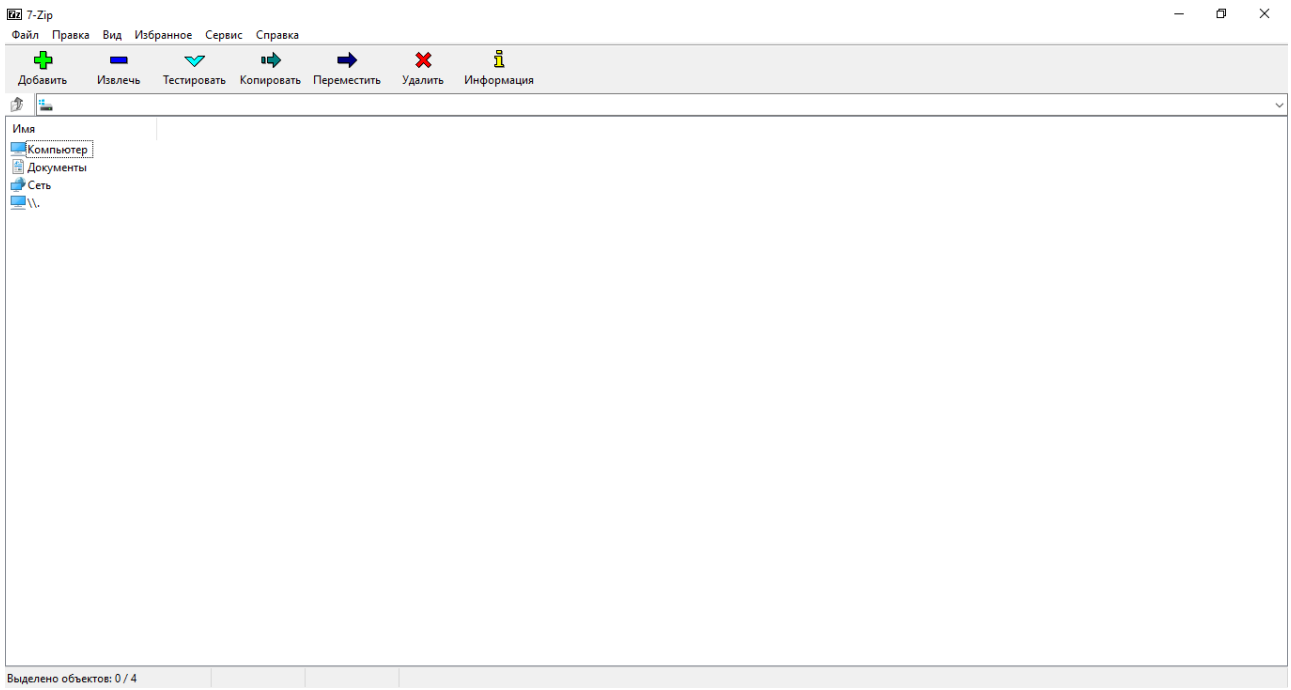


Рисунок 1.8.2.1 — Інтерфейс програми 7-Zip

1.8.3 WinRAR

WinRAR — це популярне програма для архівування файлів, яка була вперше випущена в 1995 році. Вона доступна в операційних системах Windows, Linux і

macOS і відома своїм високим ступенем стиснення та рядом функцій.

WinRAR підтримує шифрування для захисту файлів і даних, а також може створювати архіви, що саморозпаковуються, які може відкрити кожен, навіть якщо у нього не встановлено програмне забезпечення для стиснення. Ще одна чудова функція WinRAR — це здатність відновлювати пошкоджені архіви, що може бути корисним, якщо ви натрапите на пошкоджені або неповні файли.

Алгоритм WinRAR використовує комбінацію кількох різних методів стиснення, включаючи стиснення словника, стиснення ковзного вікна та ентропійне кодування. Ці методи поєднуються в унікальний спосіб для створення алгоритму, який є дуже ефективним для стиснення широкого спектру типів даних.

WinRAR інтегрується з Провідником Windows, дозволяючи легко створювати архіви та розпаковувати їх.

WinRAR є потужним і надійним інструментом архівування, якому довіряють мільйони користувачів у всьому світі. Хоча він не є безкоштовним у використанні, він пропонує безкоштовний пробний період і низку варіантів ціноутворення відповідно до різних потреб і бюджету.

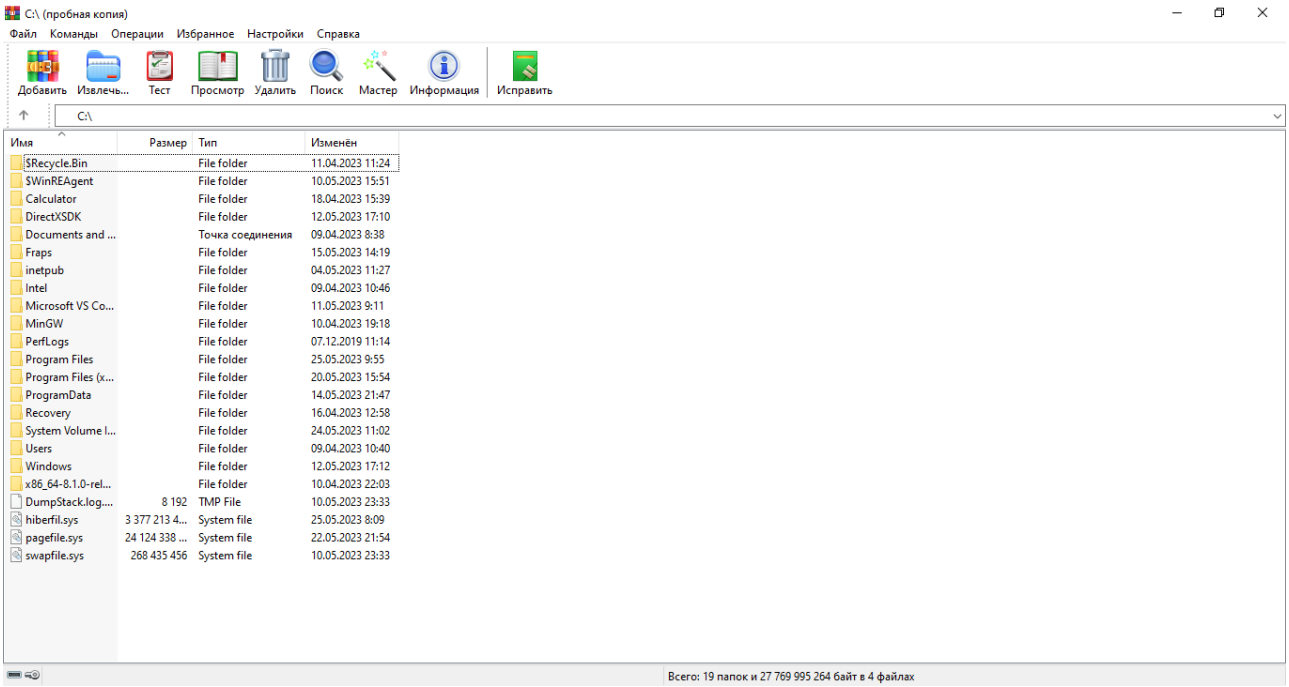


Рисунок 1.8.3.1 — Інтерфейс програми WinRAR

1.10 Таблиця порівняння програм-архіваторів

Таблиця 1.10.1 — Порівняння програм-архіваторів

	WinZIP	7-Zip	WinRAR	ZAPI Archiver
Портативність (не треба інсталювати додаток)	-	-	-	+
Висока швидкість роботи (менше 2.5 секунд на кодування або декодування)	-	+	-	+
Простий інтерфейс (невелика кількість команд, або інтеграція в оболонку ОС)	+	-	-	+
Невеликий розмір архіватора (менше 15 MiB)	-	+	+	+
Час витрачений на кодування (файл “enwik8”)	~3.5 с	~4 с	~3 с	~2.5 с

2 РОЗРОБКА ПРОГРАМИ-АРХІВАТОРА

2.1 Програмні засоби реалізації

2.1.1 Мова програмування C++

C++ — це мова програмування загального призначення, яка вперше була розроблена у 1985 році. Це популярна мова, яка широко використовується програмістами для різноманітних програм, включаючи системне програмне забезпечення, розробку ігор тощо.

Однією з причин такої популярності C++ є те, що це дуже потужна мова, яка дозволяє програмістам писати високопродуктивний код. C++ є скомпільованою мовою, що означає, що код перекладається на машинну мову перед виконанням. Це дозволяє коду C++ працювати дуже швидко, що робить його ідеальним для додатків, які вимагають високої продуктивності, таких як відеоігри або складні наукові симуляції.

C++ є популярною мовою, оскільки вона швидка, гнучка та широко підтримується. Хоча це може бути складніше для вивчення, ніж деякі інші мови програмування, це цінний інструмент для програмістів, яким потрібно писати високопродуктивний код або працювати над складними проектами.

C++ — потужна та універсальна мова програмування, яка широко використовується в алгоритмах і програмах стиснення даних. C++ особливо добре підходить для стиснення даних завдяки високій продуктивності, управлінню пам’яттю та підтримці конструкцій низькорівневого програмування.

C++ також широко використовується для розробки бібліотек стиснення та інструментів, таких як zlib і gzip. Ці бібліотеки забезпечують C++ реалізації популярних алгоритмів стиснення, а також API для стиснення та розпакування даних у програмах C++.

2.1.2 Редактор коду Visual Studio Code

Visual Studio Code, або VS Code — це безкоштовний редактор коду з відкритим кодом, який широко використовується програмістами для різноманітних мов програмування та платформ. Він був вперше випущений у 2015 році і швидко став одним із найпопулярніших доступних редакторів коду.

Однією з причин, чому VS Code люблять програмісти, є його розширюваність. Він має велику й активну спільноту розробників, які створюють і підтримують розширення для широкого діапазону мов програмування, фреймворків та інструментів. Це дозволяє розробникам налаштовувати середовище редагування відповідно до своїх потреб і вподобань.

Ще однією причиною популярності VS Code є його зручний інтерфейс. Він має простий та інтуїтивно зрозумілий інтерфейс, який полегшує розробникам навігацію та використання.

VS Code також має високу продуктивність. Він має просту архітектуру, яка дозволяє йому працювати швидко та ефективно навіть у системах з обмеженими ресурсами. Це робить його ідеальним вибором для розробників, яким потрібно працювати над великими складними проектами або яким потрібно швидко й легко перемикатися між кількома проектами.

2.1.3 Компілятор MinGW

MinGW (скорочення від Minimalist GNU for Windows) — це набір інструментів і бібліотек, які дозволяють розробникам створювати програми під операційну систему Windows за допомогою засобів розробки GNU.

Однією з ключових переваг MinGW є те, що він дозволяє розробникам використовувати ті самі інструменти та бібліотеки, які доступні в Linux або інших Unix-подібних системах, для створення програм для Windows. Це може бути особливо корисним для розробників, які вже знайомі з інструментами розробки GNU і хочуть використовувати їх для створення програм Windows.

2.1.4 Система збирання CMake

CMake — це кросплатформна система збирання з відкритим кодом, яка використовується для керування процесом збирання проектів програмного забезпечення. Він розроблений, щоб полегшити створення програмного забезпечення на багатьох платформах, включаючи Windows, macOS, Linux і Unix-подібних системах [4].

Однією з ключових переваг CMake є те, що він надає мову конфігурації, яку можна використовувати для створення файлів збірки.

CMake також підтримує широкий спектр мов програмування, включаючи C, C++, Fortran та багато інших. Це робить його універсальним інструментом, який можна використовувати для створення широкого діапазону програмних проектів.

Прикладами крупних проектів, які використовують CMake є Blender (потужне програмне забезпечення для створення 3D оточення), OpenCV (бібліотека комп’ютерного бачення та машинного навчання), VTK (бібліотека програмного забезпечення з відкритим кодом для візуалізації та обробки даних).

Отже, CMake є популярною та широко використовуваною системою збирання, яка використовується в багатьох великих і складних проектах програмного забезпечення в різних галузях, що робить його важливим інструментом для розробників програмного забезпечення та інженерів.

2.2 Створення програми

2.2.1 Проектування програми

В ході аналізу основних вимог користувачів до програм-архіваторів було вирішено реалізувати мінімальні потреби щодо програми-архіватора, а саме:

- Компіляція під платформу Windows у вигляді консольного додатку;

- Портативність (відсутність необхідності у інсталялії програми за допомогою стороннього програмного забезпечення);
- Невеликий розмір програми (менше 1 МіБ);
- Створення файлових архівів із усіх файлів вказаної директорії та кодування усіх файлів спочатку алгоритмом LZ4, після чого закодувати методом Хаффмена;
- Розпакування усіх файлів програми-архіватора у вказану директорію файлової системи та декодування кожного файлу спочатку алгоритмом LZ4, після чого методом Хаффмена;

2.2.2 Створення проекту у Visual Studio Code

Першим кроком у створенні програми-архіватора є створення проекту у редакторі коду Visual Studio Code.

Проектом у Visual Studio Code називають набір файлів і папок, організованих у єдину структуру. Проекти у Visual Studio Code зазвичай використовуються для групування пов’язаних файлів разом для певної програми чи завдання.

Для створення проекту, необхідно створити нову директорію будь-де у файловій системі, запустити Visual Studio Code та обрати “File → Open Folder”, вказавши шлях до створенної директорії.

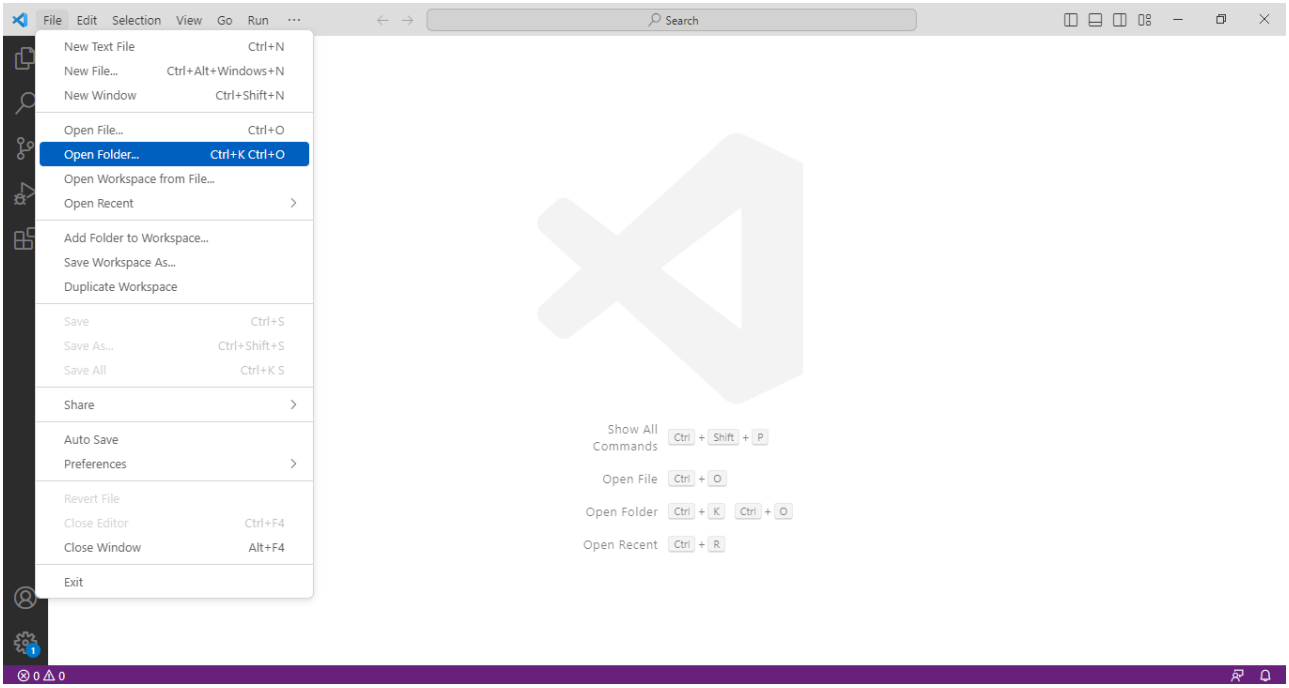


Рисунок 2.2.2.1 — Відкриття папки файлової системи для створення проекту Visual Studio Code

2.2.3 Інтерфейс програми

Користувачі можуть взаємодіяти з додатком через консоль операційної системи Windows за допомогою команд. Даний тип взаємодії був обраний з причин того, що консольний додаток простіший у використанні та займає менше місця на жорсткому диску або будь-якому накопичувачі.

Після запуску виконуваного файла програми-архіватора через консоль, відбувається вивід тексту з назвою програми-архіватора, його версією та інструкцій щодо використання програми.

```
149     std::cout << std::endl;
150     std::cout << "    ZAPI Archiver 1.0.0" << std::endl;
151     std::cout << "    Copyright (c) 2023, dmytrii4real@gmail.com" << std::endl;
152     std::cout << std::endl;
153     std::cout << "    Usage: [option] [folder_path] [archive_path]" << std::endl;
154     std::cout << "    option = [p] to pack, [x] to extract" << std::endl;
155     std::cout << "    folder_path, archive_path = full path to directory/file" << std::endl;
156     std::cout << std::endl;
```

Рисунок 2.2.3.1 — Вивід тексту після запуску програми

Користувач може вводити дві реалізовані команди, згідно з інструкцією до їх використання.

Таблиця 2.2.3.1 — Реалізовані команди

Назва	Опис
zapiapp p <folder_path> <archive_file_path>	Створення архіву <archive_file_path> з усіх файлів директорії <folder_path> (Шляхи до файлу та директорії слід вводити без знаків ‘<’ та ‘>’)
zapiapp x <folder_path> <archive_file_path>	Розпакування усіх файлів архіву <archive_file_path> до директорії <folder_path> (Шляхи до файлу та директорії слід вводити без знаків ‘<’ та ‘>’)

```
158     if (argc < 4 || (argv[1][0] != 'p' && argv[1][0] != 'x'))
159     {
160         std::cout << "    Error: Incorrect arguments!" << std::endl;
161
162         return 0;
163     }
164
165     if (argv[1][0] == 'p')
166     {
167         std::string folderPath = std::string(&argv[2][0]);
168         std::string archivePath = std::string(&argv[3][0]);
169         zapi::PackArchive pack(folderPath, archivePath);
170     }
171     else if (argv[1][0] == 'x')
172     {
173         std::string folderPath = std::string(&argv[2][0]);
174         std::string archivePath = std::string(&argv[3][0]);
175         zapi::UnpackArchive unpack(folderPath, archivePath);
176     }
```

Рисунок 2.2.3.2 — Приклад обробки команд програмою-архіватором

На рисунку 2.2.3.2 вказано обробку команди програмою-архіватором.

Рядки з 158 по 164 перевіряють аргументи командного рядка, передані програмі, і визначає, чи є перший аргумент “p” або “x”. Якщо перший аргумент не є “p” або “x”, або програмі передано менше 4 аргументів, вона виводить повідомлення про помилку на консоль і повертає 0.

“argc”: ціла змінна, яка представляє кількість аргументів, переданих

програмі з командного рядка.

“argv”: масив рядків, який містить фактичні аргументи, передані програмі з командного рядка. У цьому коді ми перевіряємо перший символ другого аргументу (argv[1][0]), щоб побачити, чи є він “p” чи “x”.

З рядка 165 по рядок 176 виконується розгалудження та перевіряється перший символ другого аргументу, переданого програмі з командного рядка, і на основі цього або пакуються, або розпаковуються файли архіву. Якщо першим символом другого аргументу є “p”, він запаковує архів. Якщо це “x”, він розпаковує архів.

“std::string folderPath”: рядкова змінна, яка містить шлях до папки, яку потрібно запакувати або розпакувати.

“std::string archivePath”: рядкова змінна, яка містить шлях до архівного файлу, який потрібно створити або розпакувати.

“zapi::PackArchive pack”: об’єкт класу PackArchive, який використовується для пакування архіву.

“zapi::UnpackArchive unpack”: об’єкт класу UnpackArchive, який використовується для розпакування архіву.

2.2.4 Реалізація кодеку

Кодек — це програмне забезпечення, яке використовується для кодування або декодування цифрових даних. Термін “кодек” в скороченні означає “кодер-декодер”.

Програма-архіватор кодує та декодує дані алгоритмом LZ4 та ентропійним кодуванням Хаффмена.

Файл “source/zapi.hpp” містить інтефейс кодеку (рисунок 2.2.4.1).

```
48  class Codec_Interface
49  {
50      public:
51          Codec_Interface() {}
52          ~Codec_Interface() {}
53
54          virtual void EncodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer) = 0;
55          virtual void DecodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer) = 0;
56  };
```

Рисунок 2.2.4.1 — Інтерфейс кодеку програми-архіватора

В програмі-архіваторі кодек реалізований у вигляді кодування алгоритмом LZ4 з наступним ентропійним кодуванням Хаффмена.

На рисунку 2.2.4.2 зображено метод “EncodeBuffer” класу “CodecLZ77WithEntropy” який стискає дані за допомогою алгоритму LZ4 (рядки 205-211). Він створює тимчасовий буфер під назвою “lz4TempBuffer” розміром 128 МіБ і ініціалізує його як порожній. Вхідні дані зчитуються в буфер “inputBuffer”. Потім викликається функція “LZ4_compress_default()” з буфером вхідних даних, тимчасовим буфером і розміром буфера вхідних даних.
Функція стискає вхідні дані та зберігає стиснуті дані у тимчасовому буфері.

Після цього код оновлює розмір стиснутих даних у тимчасовому буфері до “lz4CodedByteSize” за допомогою функції “GetSizeRef()”.

Наступним кроком є кодування методом Хаффмена (рядки 213-219). Код виконує кодування Хаффмена над стисненими даними за допомогою функції “HuffmanEncodeData()”.

Код створює тимчасовий буфер під назвою “huffmanTempBuffer” розміром 128 мегабайт і ініціалізує його як порожній. Результати попереднього кроку стиснення LZ4 зберігаються в буфері під назвою "lz4TempBuffer". Функція “HuffmanEncodeData()” викликається з буфером введення, що є стиснутими даними LZ4 у “lz4TempBuffer”, а буфером виведення — “huffmanTempBuffer”. Ця функція стискає дані за допомогою алгоритму кодування Хаффмана та зберігає стиснуті дані в “huffmanTempBuffer”.

```
202  void zapi::CodecLZ77WithEntropy::EncodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer)
203  {
204      {
205          // LZ4 coding
206          size_t lz4TempBufferByteSize = 128 * 1024 * 1024;
207          Buffer lz4TempBuffer(lz4TempBufferByteSize, nullptr);
208          size_t lz4CodedByteSize = LZ4_compress_default((const char*)inputBuffer->GetData(),
209              (char*)lz4TempBuffer.GetData(), inputBuffer->GetSizeRef(), inputBuffer->GetSizeRef());
210          lz4TempBuffer.GetSizeRef() = lz4CodedByteSize;
211
212          {
213              // Huffman coding
214              size_t huffmanTempBufferByteSize = 128 * 1024 * 1024;
215              Buffer huffmanTempBuffer(huffmanTempBufferByteSize, nullptr);
216              HuffmanEncodeData((uint8_t*)lz4TempBuffer.GetData(),
217                  (uint8_t*)huffmanTempBuffer.GetData(), lz4TempBuffer.GetSizeRef(), huffmanTempBufferByteSize);
218              huffmanTempBuffer.GetSizeRef() = huffmanTempBufferByteSize;
219
220              // Copy data to output buffer
221              memcpy(outputBuffer->GetData(), huffmanTempBuffer.GetData(), huffmanTempBufferByteSize);
222              outputBuffer->GetSizeRef() = huffmanTempBufferByteSize;
223          }
224      }
225  }
```

Рисунок 2.2.4.2 — Реалізація методу кодування

Аналогічно до кодування виконується декодування, лише з оберненими функціями — спочатку вхідний дані декодуються методом Хаффмена, а лише потім алгоритмом LZ4 (рисунок 2.2.4.3).

```
227  void zapi::CodecLZ77WithEntropy::DecodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer)
228  {
229      {
230          // Huffman decoding
231          size_t huffmanTempBufferByteSize = 128 * 1024 * 1024;
232          Buffer huffmanTempBuffer(huffmanTempBufferByteSize, nullptr);
233          HuffmanDecodeData((uint8_t*)inputBuffer->GetData(),
234              (uint8_t*)huffmanTempBuffer.GetData(), inputBuffer->GetSizeRef(), huffmanTempBufferByteSize);
235          huffmanTempBuffer.GetSizeRef() = huffmanTempBufferByteSize;
236
237          {
238              // LZ4 decoding
239              size_t lz4TempBufferByteSize = 128 * 1024 * 1024;
240              Buffer lz4TempBuffer(lz4TempBufferByteSize, nullptr);
241              size_t lz4DecodedByteSize = LZ4_decompress_safe((const char*)huffmanTempBuffer.GetData(),
242                  (char*)lz4TempBuffer.GetData(), huffmanTempBuffer.GetSizeRef(), lz4TempBufferByteSize);
243              lz4TempBuffer.GetSizeRef() = lz4DecodedByteSize;
244
245              // Copy data to output buffer
246              memcpy(outputBuffer->GetData(), lz4TempBuffer.GetData(), lz4DecodedByteSize);
247              outputBuffer->GetSizeRef() = lz4DecodedByteSize;
248          }
249      }
250  }
```

Рисунок 2.2.4.3 — Реалізація методу декодування

Кодек програми-архіватора працює з буферами, тому варто розглянути реалізацію буфера в даній роботі.

Буфер — це безперервний блок пам’яті, призначений для зберігання певної кількості даних. Розмір буфера зазвичай вказується під час його створення, а дані зчитуються або записуються в буфер порціями.

Буфер визначений в файлі “source/zapi.h” та реалізований у файлі “source/zapi.cpp”.

```
6   class Buffer
7   {
8       public:
9           explicit Buffer(size_t byteSize, void* data);
10          ~Buffer();
11
12          size_t& GetSizeRef() { return _byteSize; }
13          void* GetData() { return _data; }
14
15      private:
16          size_t _byteSize;
17          void* _data;
18  };
```

Рисунок 2.2.4.1 — Визначення класу Buffer

Код на рисунку 2.2.4.1 визначає клас під назвою “Buffer”. Клас “Buffer” має дві приватні змінні: “_byteSize” і “_data”, які використовуються для зберігання розміру буфера (у байтах) і вказівника на дані в буфері відповідно.

Конструктор класу “Buffer” приймає два параметри: “byteSize” і “data”. “byteSize” вказує розмір буфера, який потрібно виділити (у байтах), а “data” є вказівником на пам’ять, яка вже була виділена для буфера.

Функція “GetSizeRef()” повертає посилання на розмір буфера. Посилання на розмір буфера повертається, щоб його можна було змінити іншою функцією чи об’єктом, якщо необхідно.

Функція “GetData()” повертає вказівник на дані в буфері. Ця функція використовується для отримання даних, що зберігаються в буфері.

Деструктор класу "Buffer" звільняє пам'ять, яку використовує буфер, коли

вона більше не потрібна. Це робиться для запобігання витоку пам'яті та забезпечення ефективного використання пам'яті.

```
114  zapi::Buffer::Buffer(size_t byteSize, void* data)
115  {
116      if (byteSize != 0)
117      {
118          _byteSize = byteSize;
119          _data = malloc(byteSize);
120
121          if (data != nullptr) {
122              memcpy(_data, data, byteSize);
123          }
124      }
125      else
126      {
127          _byteSize = 0;
128          _data = nullptr;
129      }
130  }
131
132  zapi::Buffer::~Buffer()
133  {
134      if (_data != nullptr) {
135          free(_data);
136      }
137  }
```

Рисунок 2.2.4.2 — Конструктор і деструктор класу Buffer

Код на рисунку 2.2.4.2 визначає конструктор і деструктор класу “Buffer”.

Конструктор класу “Buffer” приймає два параметри: “byteSize” і “data”. “byteSize” вказує розмір буфера, який потрібно виділити (у байтах), а “data” — це вказівник на пам’ять, яка вже була виділена для буфера.

Конструктор спочатку перевіряє, чи не дорівнює параметр “byteSize” нулю (рядок 116). Якщо він не дорівнює нулю, він виділяє пам'ять для буфера за допомогою функції “malloc()”. Потім він ініціалізує змінну-член “_byteSize” значенням “byteSize”. Якщо параметр “data” не є нульовим покажчиком, конструктор копіює дані, на які вказує “data”, у буфер за допомогою функції “memcpy()” (рядки 121-123).

Якщо параметр “byteSize” дорівнює нулю, конструктор встановлює значення змінної-члена “_byteSize” рівним нулю (рядок 127), а значення змінної-члена “_data” — нульовим покажчиком (рядок 128).

Деструктор класу “Buffer” (рядок 132) перевіряє, чи змінна-член “_data” не є нульовим покажчиком (рядок 134). Якщо це не нульовий покажчик, він звільняє пам’ять, яку використовує буфер, за допомогою функції “free()” (рядок 135).

Ці функції конструктора та деструктора гарантують, що буферна пам’ять правильно розподіляється та звільняється, а також що будь-які дані, що зберігаються в буфері, копіюються або видаляються за необхідності.

2.2.5 Створення файлового архіву

За створення файлового архіву відповідає клас “PackArchive”, який визначений у файлі “source/zapi.hpp”, а його конструктор у файлі “source/zapi.cpp”.

Весь функціонал інкапсульований у конструктор класу “PackArchive” (рисунок 2.2.5.1), що приймає на вхід два аргументи: шлях до директорії у файловій системі, усі файли якої мають бути архівовані, та шлях до файлу архіву, куди будуть збережені стиснуті файли.

Цей клас також має два відкритих методи та конструктор.

Перший публічний метод, “GetFolderPathRef()”, повертає посилання на змінну “_folderPath”, щоб до неї можна було отримати доступ поза класом.

Другий публічний метод, “GetArchivePathRef()”, повертає посилання на змінну “_archivePath”, щоб до неї можна було отримати доступ поза класом.

```
20  class PackArchive
21  {
22      public:
23          explicit PackArchive(std::string folderPath, std::string archivePath);
24          ~PackArchive();
25
26          std::string& GetFolderPathRef() { return _folderPath; }
27          std::string& GetArchivePathRef() { return _archivePath; }
28
29      private:
30          std::string _folderPath;
31          std::string _archivePath;
32  };
```

Рисунок 2.2.5.1 — Визначення класу PackArchive

```
176  zapi::PackArchive::PackArchive(std::string folderPath, std::string archivePath)
177  {
178      _folderPath = folderPath, _archivePath = archivePath;
179      std::string folderName, reversedFolderName;
180
181      int i = _folderPath.size() - 1;
182      while (_folderPath[i] != '/') {
183          char appendStr[2] = { _folderPath[i], 0 };
184          reversedFolderName.append(&appendStr[0]);
185          i -= 1;
186      }
187
188      i = reversedFolderName.size() - 1;
189      folderName.reserve(reversedFolderName.size());
190      while (i > -1) {
191          char appendStr[2] = { reversedFolderName[i], 0 };
192          folderName.append(&appendStr[0]);
193          i -= 1;
194      }
195
196      unsigned fileCount = 0;
197      std::ofstream archiveFileStream(_archivePath, std::ios::out | std::ios::binary);
198      if (archiveFileStream.fail()) { std::cout << "    Error: Archive file not found!" << std::endl; return; }
199
200      archiveFileStream.write((const char*)&fileCount, sizeof(unsigned));
201      ReadDirectories(archiveFileStream, _folderPath, folderName);
202      archiveFileStream.close();
203
204      std::cout << std::endl << "    Info: Archive files were packed successfully!" << std::endl;
205  }
```

Рисунок 2.2.5.2 — Визначення конструктора класу “PackArchive”

На рисунку 2.2.5.2, з рядка 178 по рядок 195 відбувається пошук назви директорії з заданого шляху до файлу. Шлях до файлу зберігається в рядковій змінній “_folderPath”.

Код спочатку ініціалізує змінну лічильника і індексом останнього символу в рядку “_folderPath”. Потім він перебирає рядок у зворотному напрямку, додаючи кожен символ до нової рядкової змінної “reversedFolderName”. Це триває, доки цикл не досягне першої косої риски “/” (рядки 181-186).

Після завершення циклу код ініціалізує “i” до останнього індексу “reversedFolderName” і резервує місце в новій рядковій змінній “folderName” для імені директорії. Потім код перебирає перевернутий рядок імені директорії від кінця до початку, додаючи кожен символ до “folderName”. Це фактично змінює

порядок символів у перевернутому рядку імені директорії, що призводить до правильної назви директорії.

Рядок 197 відкриває файловий потік для запису за вказаним шляхом до файлу архіву “archivePath”. Рядок 198 перевіряє чи файл вдало відкрився.

Рядок “archiveFileStream.write((const char*)fileCount, sizeof(unsigned));” відповідає за запис кількості файлів в архіві та в даному випадку записує число нуль і більше необхідний для того, щоб сдвинути курсор потоку далі на кількість байт рівній розміру типу “unsigned int”, та у разі якщо поданий новий файл для архівації, пересунути курсор на початок запису у потік та оновити значення кількості файлів.

Рекурсивна функція “ReadDirectories”, зображена нижче на рисунку 2.2.5.3, приймає на вхід потік для запису, шлях до директорії, яка має бути архівована, та назву останньої папки зі шляху до директорії.

Робота функції “ReadDirectories” заключається в тому, щоб пройти усіма файлами, які містяться у вказаній директорії та її піддиректоріях і додати усі ці файли до архіву.

Рядки 110 та 111 використовують Windows API для пошуку першого файлу в заданій директорії. Вони ініціалізують структуру WIN32_FIND_DATAA, яка містить інформацію про файл, знайдений функцією. Потім викликається функція FindFirstFileA з двома параметрами: рядок, що містить шлях до директорії, у якій потрібно виконати пошук (який поєднується з “*” для пошуку всіх файлів у каталозі), і вказівник на структуру WIN32_FIND_DATAA для отримання інформації про перший знайдений файл у директорії. Функція повертає дескриптор пошуку, який зберігається в змінній HANDLE hFind.

Починаючи з рядка 113 виконується перевірка, чи файл, знайдений у директорії, є самою директорією чи ні. Якщо файл є директорією, код формує новий шлях до директорії та локальний шлях до директорії за допомогою попередньо знайденої директорії, всередині блоку “if”, після чого викликає функцію “ReadDirectories” вже з новими аргументами. Якщо файл не є

директорією тоді всередині блоку “else”, рядки 126 по 129 зберігають курсор потоку, після чого здвигають курсор в початок та збільшують лічильник файлів архіву на значення “1”, після чого повертаються до збереженої позиції потоку. Далі викликається функція “PackFile”, яка додає файл до архіву.

Цикл (рядок 135) перебирає всі файли в директорії, виконуючи одну з операцій залежно від того, чи є файл директорією чи ні. Коли в директорії більше немає файлів для пошуку, цикл завершується. Потім він викликає функцію “FindClose”, щоб закрити дескриптор пошуку та звільнити будь-які пов’язані з ним системні ресурси.

```
108  void ReadDirectories(std::ofstream& archiveFileStream, const std::string& path, const std::string& localPath)
109  {
110      WIN32_FIND_DATAA findData;
111      HANDLE hFind = FindFirstFileA((path + "\\*").c_str(), &findData);
112
113      if (hFind != INVALID_HANDLE_VALUE) {
114          do {
115              if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
116                  if (strcmp(findData.cFileName, ".") != 0 && strcmp(findData.cFileName, "..") != 0) {
117                      std::string directoryPath = path + "\\" + findData.cFileName;
118                      std::string newLocalPath = localPath + "\\" + findData.cFileName;
119                      ReadDirectories(archiveFileStream, directoryPath, newLocalPath);
120                  }
121              } else {
122                  static unsigned fileCount = 0; fileCount += 1;
123
124                  std::string newLocalPath = localPath + "\\" + findData.cFileName;
125
126                  auto streamPos = archiveFileStream.tellp();
127                  archiveFileStream.seekp(0, std::ios::beg);
128                  archiveFileStream.write((const char*)&fileCount, sizeof(unsigned));
129                  archiveFileStream.seekp(streamPos);
130
131                  PackFile(archiveFileStream, std::string(path + "\\" + findData.cFileName).c_str(), newLocalPath.c_str());
132
133                  std::cout << "    Packed file: " << newLocalPath << std::endl;
134              }
135          } while (FindNextFileA(hFind, &findData));
136          FindClose(hFind);
137      }
138  }
```

Рисунок 2.2.5.3 — Визначення функції ReadDirectories

Функція “PackFile” (рисунок 2.2.5.4), спочатку зчитує вміст файлу в пам’ять (рядки 14-20). Код спочатку створює вхідний файловий потік, передаючи конструктору шлях до файлу та двійковий прапор введення (рядок 14). Потім код шукає кінець файлу за допомогою методу “seekg()”, який повертає розмір файлу в байтах за допомогою методу “tellg()”. Під файл виділяється кількість байт, що

рівна розміру файлу, з динамічної пам’яті функцією “malloc”. Після роботи з блоком пам’яті, вона вивільняється функцією “free” (рядок 34).

Код у рядках 22-25 стискає файл за допомогою алгоритму LZ77 з ентропійним кодуванням, а потім записує стислі дані у файловий архів. Код створює два буфери: один для зберігання пам’яті файлу, а інший для зберігання стиснутих даних. Пам’ять файлу зчитується в буфер “bInput”, а функція “compress.EncodeBuffer()” викликається, щоб стиснути дані з буфера “bInput” і помістити їх у буфер “bOutput”. Потім код записує розмір стиснених даних, локальний шлях до файлу та самі стиснуті дані у двійковий файл за допомогою функції “archiveFileStream.write()”.

Локальний шлях до файлу зберігається в масиві символів “fileLocalPathStr” і копіюється з рядка “localFilePath” за допомогою функції “memcpy()”. Розмір стиснутих даних записується у двійковий файл за допомогою “sizeof(unsigned)”. Нарешті, самі стислі дані записуються у файл за допомогою “bOutput.GetData()” і “bOutput.GetSizeRef()”.

```
12  void PackFile(std::ofstream& archiveFileStream, const char* filePath, const char* localFilePath)
13  {
14      std::ifstream fileStream(filePath, std::ios::in | std::ios::binary);
15      fileStream.seekg(0, std::ios::end);
16      size_t fileByteSize = fileStream.tellg();
17      fileStream.seekg(0, std::ios::beg);
18      void* fileMemory = malloc(fileByteSize);
19      fileStream.read((char*)fileMemory, fileByteSize);
20      fileStream.close();
21
22      zapi::Buffer bInput(fileByteSize, fileMemory);
23      zapi::Buffer bOutput(100 * 1024 * 1024, nullptr);
24      zapi::CodecLZ77WithEntropy compress;
25      compress.EncodeBuffer(&bInput, &bOutput);
26
27      char fileLocalPathStr[256] = {};
28      memcpy(&fileLocalPathStr[0], &localFilePath[0], strlen(localFilePath));
29
30      archiveFileStream.write((const char*)&bOutput.GetSizeRef(), sizeof(unsigned));
31      archiveFileStream.write((const char*)&fileLocalPathStr[0], 256);
32      archiveFileStream.write((const char*)bOutput.GetData(), bOutput.GetSizeRef());
33
34      free(fileMemory);
35  }
```

Рисунок 2.2.5.4 — Визначення функції PackFile

2.2.6 Розпакування файлового архіву

Код класу “UnpackArchive”, який розпаковує архів, визначений у файлі “source/zapi.hpp”, а тіло функцій знаходиться у файлі “source/zapi.cpp”.

Весь функціонал інкапсульований у конструктор класу “UnpackArchive” (рисунок 2.2.6.1), що приймає на вхід два аргументи: шлях до директорії у файловій системі, куди файли мають бути розпаковані, та шлях до файлу архіву, усі файли якого мають бути розпаковані.

Клас має дві приватні змінні-члени “folderPath” і “archivePath” типу std::string, які зберігають шлях до папки, куди буде розпаковано архів, і шлях до файлу архіву відповідно.

Клас визначає дві загальнодоступні функції-члени “GetFolderPathRef” і “GetArchivePathRef”, які повертають посилання на приватні змінні-члени “folderPath” і “archivePath” відповідно. Ці функції можна використовувати для доступу до рядків шляху, що зберігаються в об’єкті “UnpackArchive”, поза класом.

```
34  class UnpackArchive
35  {
36      public:
37          explicit UnpackArchive(std::string folderPath, std::string archivePath);
38          ~UnpackArchive();
39
40          std::string& GetFolderPathRef() { return _folderPath; }
41          std::string& GetArchivePathRef() { return _archivePath; }
42
43      private:
44          std::string _folderPath;
45          std::string _archivePath;
46  };
```

Рисунок 2.2.6.1 — Визначення класу UnpackArchive

```
204  zapi::UnpackArchive::UnpackArchive(std::string folderPath, std::string archivePath)
205  {
206      _folderPath = folderPath;
207      _archivePath = archivePath;
208
209      unsigned fileCount = 0;
210      std::ifstream archiveFileStream(_archivePath, std::ios::in | std::ios::binary);
211      if (archiveFileStream.fail()) {
212          std::cout << "    Error: Archive file not found!" << std::endl;
213
214          return;
215      }
216
217      archiveFileStream.read((char*)&fileCount, sizeof(unsigned));
218      while (fileCount > 0)
219      {
220          UnpackFile(archiveFileStream, _folderPath.c_str());
221          fileCount -= 1;
222      }
223      archiveFileStream.close();
224
225      std::cout << std::endl << "    Info: Archive files were extracted successfully!" << std::endl;
226  }
```

Рисунок 2.2.6.2 — Визначення конструктора класу UnpackArchive

На рисунку 2.2.6.2 зображено код, який читає файл архіву, визначений змінною “_archivePath”, що містить інформацію про файли, які потрібно розпакувати. Після успішного відкриття файлу код зчитує кількість файлів в архіві з початку файлу, а потім переходить до розпакування кожного файлу за допомогою функції “UnpackFile()”, яка розпаковує кожен файл з архіву та зберігає його у вказану папку “_folderPath”. Код використовує цикл “while”, щоб відстежувати, скільки файлів залишилося розпакувати, зменшуючи змінну “fileCount” на одиницю в кожній ітерації, доки всі файли не будуть розпаковані. Якщо архівний файл не відкривається, код видає повідомлення про помилку та завершує роботу функції.

```
41  void UnpackFile(std::ifstream& archiveFileStream, const char* folderPath)
42  {
43      std::string filePath(folderPath); unsigned fileByteSize = 0; char fileLocalPathStr[256] = {}; void* fileMemory = nullptr;
44
45      archiveFileStream.read((char*)&fileByteSize, sizeof(unsigned));
46      archiveFileStream.read((char*)&fileLocalPathStr[0], 256);
47      fileMemory = malloc(fileByteSize);
48      archiveFileStream.read((char*)fileMemory, fileByteSize);
49
50      filePath = filePath + '\\' + fileLocalPathStr;
51      std::string localFolderName;
52      char appendStr[2] = { filePath[0], 0 };
53      localFolderName.append(&appendStr[0]);
54
55      for (int i = 1; i < filePath.size();) {
56          char appendStr[2] = { filePath[i], 0 };
57          localFolderName.append(&appendStr[0]);
58          if (filePath[i - 1] == '/' || filePath[i - 1] == '\\') {
59              i += 1;
60              while (filePath[i] != '/' && filePath[i] != '\\' && i < filePath.size()) {
61                  char appendStr[2] = { filePath[i], 0 };
62                  localFolderName.append(&appendStr[0]);
63                  i += 1;
64              }
65              if (i != filePath.size()) { CreateDirectoryA(localFolderName.c_str(), NULL); } else { break; }
66          } else {
67              i += 1;
68          }
69      }
70
```

Рисунок 2.2.6.3 — Перша частина тіла функції UnpackFile

Перша частина функції “UnpackFile” (рисунок 2.2.6.3) спочатку читає файл із файлу архіву (рядки 43-53), визначеного змінною “archiveFileStream”. Код зчитує розмір файлу в байтах і шлях до файлу з файлу архіву, а потім виділяє пам’ять для файлу за допомогою функції “malloc()”. Змінна fileByteSize зберігає розмір файлу, тоді як змінна “fileLocalPathStr” зберігає шлях до файлу. Функція “archiveFileStream.read()” використовується для читання даних з архівного файлу у вид'ілений блок пам’яті. Нарешті, дані файлу доступні в змінній “fileMemory”, за допомогою якої можна записати файл на диск або виконати над ним інші операції.

Наступни кроком (рядки 50-53) виконується створення повного шляху до файлу, який розпаковується з архіву. Код об’єднує шлях до папки “filePath” із шляхом до файлу “fileLocalPathStr”, вставляючи символ зворотної косої риски “\” між ними, щоб сформувати повний шлях. Отриманий шлях до файлу зберігається у змінній “filePath”. Після цього код створює рядок “localFolderName”, що містить назву папки, де знаходиться файл. Ім’я папки отримується шляхом взяття першого символу змінної “filePath” і збереження його в масиві символів “appendStr”. Потім

створюється рядок “localFolderName” шляхом додавання до нього першого символу “filePath”.

Далі (рядки 55-69) виконується код, який створює каталоги на основі батьківсько-дочірніх зв’язків, визначених скісною рискою в шляху до файлу. Код перебирає кожен символ у шляху до файлу, додаючи кожен символ до рядка “localFolderName”. Якщо поточний символ є косою рискою “/”, цикл продовжується по шляху, доки не досягне наступної косої риски або кінця шляху. Коли він досягає кінця директорії, він створює цю директорію за допомогою функції “CreateDirectoryA”.

```
74      // Decompress file data
75      zapi::Buffer bInput(fileByteSize, fileMemory);
76      zapi::Buffer bOutput(100 * 1024 * 1024, nullptr);
77      zapi::CodecLZ77WithEntropy decompress;
78      decompress.DecodeBuffer(&bInput, &bOutput);
79
80      std::ofstream fileStream(filePath.c_str(), std::ios::out | std::ios::binary);
81      fileStream.write((const char*)bOutput.GetData(), bOutput.GetSizeRef());
82      fileStream.close();
83
84      free(fileMemory);
85
86      std::cout << "    Info: Unpacked file: " << filePath << std::endl;
87  }
```

Рисунок 2.2.6.4 — Друга частина тіла функції UnpackFile

Друга частина функції “UnpackFile” (рисунок 2.2.6.4) зчитує стислі дані з пам’яті, розпаковує їх, записує розпаковані дані у файл на диску та звільняє виділену пам’ять.

З рядка 75 по 79 код розпаковує стислий файл архіву за допомогою алгоритму LZ77 з ентропійним кодуванням, а потім зберігає розпаковані дані у файл. Код створює два буфери: один для збереження пам’яті стисненого файлу, а інший для зберігання розпакованих даних. Пам’ять стисненого файлу зчитується в буфер “bInput”, і викликається функція “decompress.DecodeBuffer()”, щоб розпакувати дані з буфера “bInput” і помістити їх у буфер “bOutput”. Після цього код записує розпаковані дані у файл. Нарешті, він звільняє пам’ять, яку використовує стиснутий файл, і друкує повідомлення про те, що файл

розпаковано.

Потім код відкриває потік файлу (рядок 88) за вказаним шляхом до файлу архіву з об’єктом “std::ofstream”. Потім він записує розпаковані дані у файл. В кінці, він закриває потік файлів і звільняє пам’ять, виділену для закодованих даних.

2.2.7 Алгоритм LZ4

Опис інтерфейсу алгоритму знаходиться в “codecs/lz4.h”, а реалізація в “codecs/lz4.c”.

LZ4 — це алгоритм стиснення даних без втрат, який використовується в програмах архівування файлів і передачі даних. Алгоритм розроблено для роботи на дуже високих швидкостях, що робить його популярним для додатків, які потребують швидкого стиснення та розпакування.

LZ4 часто використовується для архівування файлів, оскільки він може досягти високого коефіцієнта стиснення, зберігаючи високу швидкість стиснення та розпакування. Стискаючи файли за допомогою LZ4, користувачі
можуть заощадити місце на диску та скоротити час, необхідний для передачі файлів через Інтернет.

Алгоритм LZ4 використовує формат стиснення на основі блоків, де кожен блок стискається незалежно. Формат стисненого блоку починається із заголовка, який описує спосіб стиснення блоку.

Заголовок — це 32-розрядне ціле число без знаку, яке розділене на дві частини: перші 4 біти описують розмір блоку (у ступенях 2), а решта 28 бітів описують стиснений і розпакований розміри блоку. Стиснутий розмір зберігається як 32-розрядне ціле число без знаку, тоді як розпакований розмір зберігається як 16-розрядне ціле число без знаку.

Після заголовка стислий блок містить серію токенів, які представляють літерали та збіги. Літеральний маркер — це байт нестиснутих даних, які

копіюються безпосередньо у вихідний буфер. Маркер відповідності представляє послідовність байтів, які раніше бачили у вхідному буфері. Маркер відповідності містить два значення: зсув відповідної послідовності (відносно поточної позиції у вихідному буфері) і довжину відповідної послідовності.

Алгоритм LZ4 використовує ряд методів для покращення швидкості стиснення та розпакування, наприклад використання хеш-таблиці для швидкого пошуку відповідних послідовностей і використання ковзного вікна для обмеження розміру словника. Загалом, формат LZ4 розроблено таким чином, щоб бути ефективним і простим у застосуванні, що робить його популярним вибором для програм стиснення в реальному часі.

2.2.8 Метод Хаффмена

Опис інтерфейсу алгоритму знаходиться в “codecs/huffman.hpp”, а визначення функцій в “codecs/huffman.cpp”.

Алгоритм Хаффмена — це широко використовуваний алгоритм стиснення даних, який використовує кодування Хаффмена для стиснення даних. Алгоритм працює, призначаючи коротші коди символам, які часто зустрічаються, і довші коди рідкісним символам.

Алгоритм починається з обчислення частоти кожного символу у вхідних даних. Потім він будує дерево Хаффмена, багаторазово об’єднуючи два вузли з найменшими частотами в єдиний батьківський вузол. Цей процес триває, доки всі вузли не будуть об’єднані в один кореневий вузол.

Після того, як дерево Хаффмена було побудовано, кожному символу присвоюється код шляхом обходу дерева від кореня до листкового вузла, який відповідає символу. Код, присвоєний кожному символу, є послідовністю двійкових цифр уздовж шляху від кореня до кінцевого вузла.

Закодовані дані зазвичай зберігаються у вигляді послідовності бітів, причому кожен символ представлений відповідним кодом. Декодування

закодованих даних полягає в проходженні дерева Хаффмена від кореня до кінцевого вузла, використовуючи кожен біт закодованих даних, щоб визначити, чи потрібно перейти до лівого чи правого дочірнього вузла. Коли кінцевий вузол досягається, виводиться відповідний символ і обхід продовжується з кореневого вузла.

Кодування Хаффмена широко використовується в програмах для стиснення даних, оскільки воно просте, ефективне та легко реалізоване. Він може досягти високих коефіцієнтів стиснення для певних типів даних, особливо тих, що мають спотворений частотний розподіл символів.

2.2.9 Збирання проекту за допомогою CMake

Збирання проекту за допомогою CMake проходить у три етапи. Спочатку створюється і прописується конфігураційний файл CMake.

```
1   cmake_minimum_required(VERSION 3.2)
2
3   project(zapiapp)
4
5   set(CMAKE_CXX_STANDARD 11)
6   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3")
7
8   set(SOURCE_FILES
9       codecs/elz1.c
10      codecs/lz4.c
11      codecs/huffman.cpp
12      source/zapi.cpp
13      samples/main.cpp
14  )
15
16  add_executable(zapiapp ${SOURCE_FILES})
```

Рисунок 2.2.9.1 — Конфігураційний файл CMakeLists.txt

Файл CMake зображений на рисунку 2.2.9.1 визначає конфігурацію збірки для проекту C++ під назвою “zapiapp”. На початку файлу вимагається версія CMake 3.2 або вище.

Проект визначається за допомогою команди “project”, яка задає назву

проекту.

Команда “set” використовується для встановлення стандарту C++, який буде використовуватися компілятором, встановлюється на “C++11”. Конфігурація також встановлює прапори компілятора для оптимізації швидкості за допомогою “-O3”.

Список вихідних файлів, які будуть скомпільовані та пов’язані в остаточний виконуваний файл, визначається у змінній “SOURCE_FILES”. Ці файли містять файли вихідного коду C і C++ із кодеків і вихідних директорій, а також вихідний файл C++ із директорії зразків.

В кінці, команда “add_executable” використовується для створення виконуваного файлу під назвою “zapiapp” зі списку “SOURCE_FILES”. Ця команда вказує, що для компіляції файлів і зв’язування їх у кінцевий виконуваний файл слід використовувати компілятор C++.

Загалом, цей файл CMake визначає конфігурацію збірки для проекту “zapiapp”, включаючи вихідні файли, які будуть скомпільовані та зв’язані в кінцевий виконуваний файл.

Наступним кроком після створення та написання конфігураційного файлу CMake, є генерація файлів для обраного користувачем компілятора. Для генерації файлів для компілятора, необхідно відкрити програму “cmake-gui”, далі обрати “Tools → Generate”, та у новому вікні обрати компілятор, за допомогою якого буде виконуватись компіляція проеку, після чого натиснути “Finish” (зображено на рисунку 2.2.9.2).

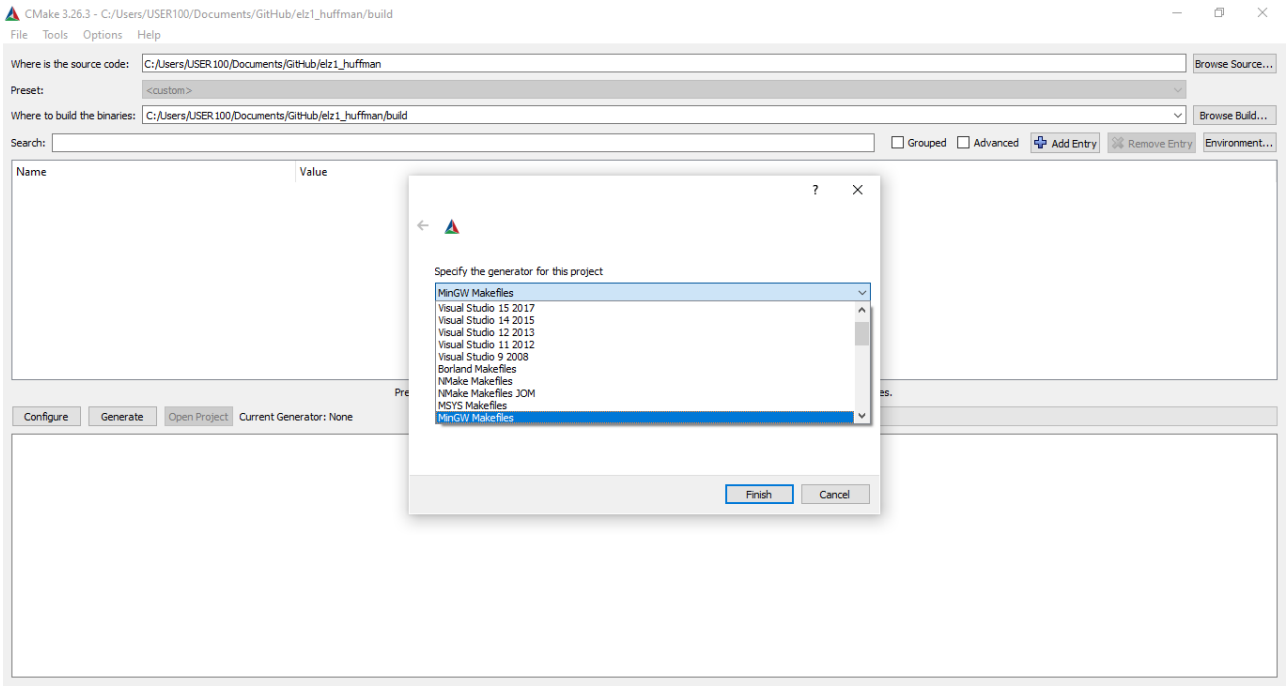


Рисунок 2.2.9.2 — Генерація файлів для обраного користувачем компілятора

Завершальним етапом збирання проекту є безпосередня компіляція програми обраним користувачем компілятором. Для побудови програми-архіватора був використаний компілятор MinGW. Для побудови необхідно відкрити консоль у директорії, куди були згенеровані файли для компілятора програмою “cmake-gui”, та вписати “mingw32-make”, ця команда скомпілює проект у вигляд виконуваного файлу, згідно з конфігурацією CMake, безпосередньо в директорію, де була відкрита консоль (рисунок 2.2.9.3).

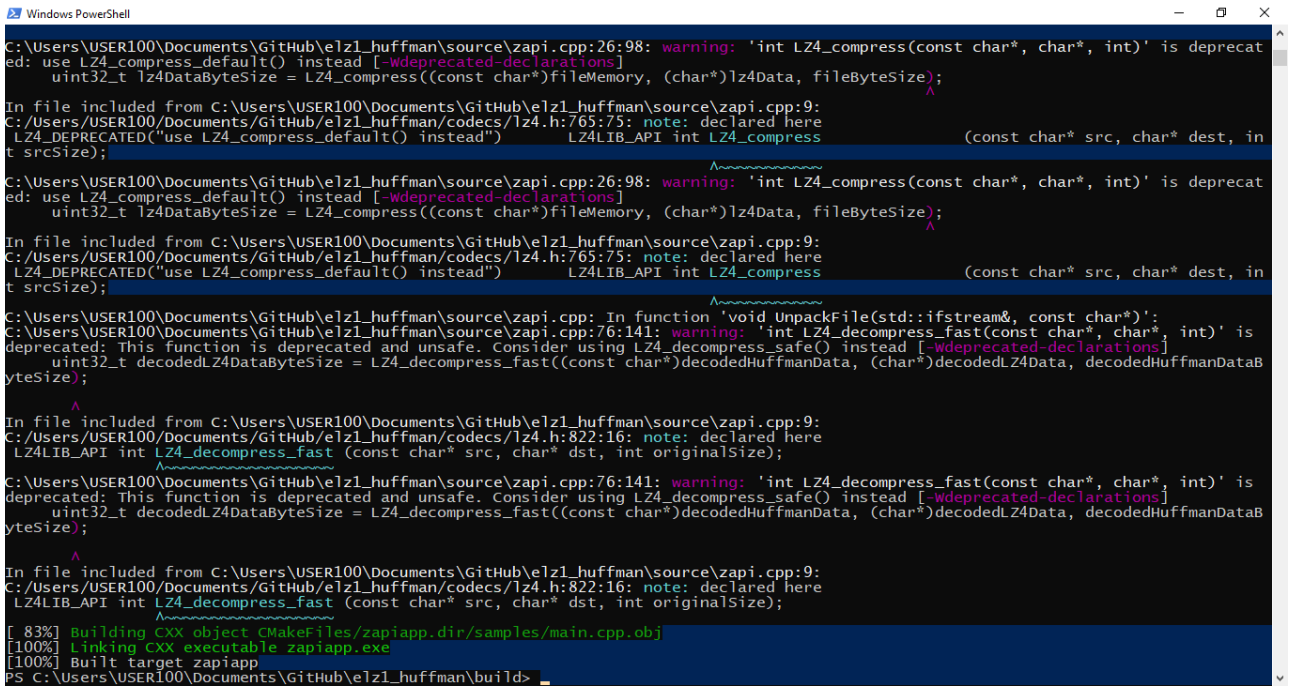


Рисунок 2.2.9.3 — Успішна компіляція проекту

3 ПРИКЛАДИ ВИКОРИСТАННЯ ТА ТЕСТУВАННЯ ПРОГРАМИ-АРХІВАТОРА

3.1 Початок роботи з програмою

Щоб розпочати роботи з програмою, необхідно відкрити консоль, це можна зробити одночасно натиснувши клавіші “Win” та “R” на клавіатурі, потім ввести в текстове поле “cmd”, після чого ввести в консоль команду “cd <шлях_до_папки_з_програмою>”, без зайвих знаків “<” та “>” (на операційній системі Windows). В результаті чого, консоль користувача перейде до директорії з програмою-архіватором.

Для того, щоб розпочати роботу, спершу треба дізнатися про те, як працює програма. Для цього необхідно ввести команду “zapiapp“, після вводу команди консоль виведе інструкції, щодо використання (рисунок 3.1.1).

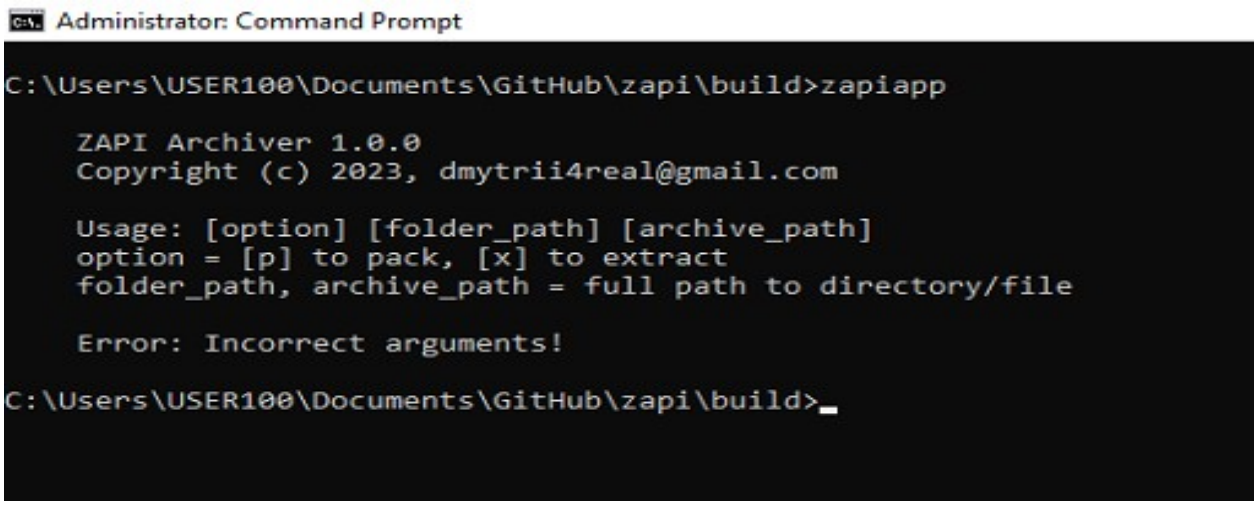


Рисунок 3.1.1 — Інструкції до використання програми-архіватора

3.2 Робота з програмою

Користувач може вводити команди, які зображені на таблиці 3.2.1.

Таблиця 3.2.1 — Команди програми-архіватора для введеня користувачем

Назва	Опис
zapiapp p <folder_path> <archive_file_path>	Створення архіву <archive_file_path> з усіх файлів директорії <folder_path> (Шляхи до файлу та директорії слід вводити без знаків ‘<’ та ‘>’)
zapiapp x <folder_path> <archive_file_path>	Розпакування усіх файлів архіву <archive_file_path> до директорії <folder_path> (Шляхи до файлу та директорії слід вводити без знаків ‘<’ та ‘>’)

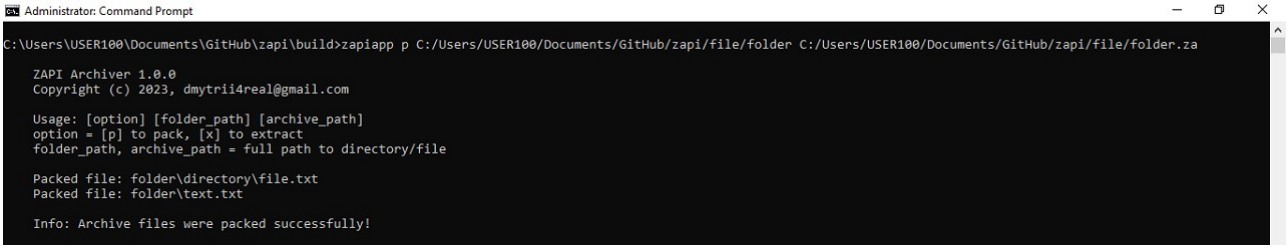


Рисунок 3.2.1 — Успішне виконання команди створення файлового архіву

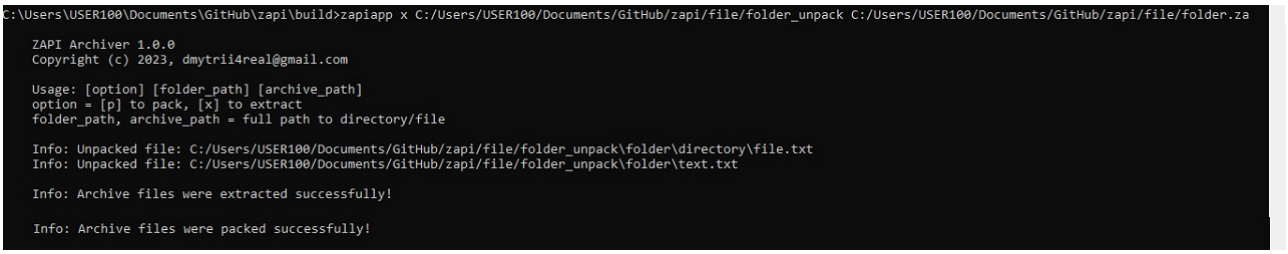


Рисунок 3.2.2 — Успішне виконання команди розпаковки файлового архіву

3.3 Фіксування результатів роботи програми

Для фіксування результатіва роботи програми було використано бібліотеку C++ “chrono” та показники програми “explorer” операційної системи Windows.

Для фіксування часу роботи програми, у програмний код було додано рядки, які зображено на рисунку 3.3.1.

Код з рядка 170 по 174 вимірює час, потрібний для створення стисненого архіву директорії за допомогою конструктору “zapi::PackArchive”.

Код спочатку ініціалізує час початку, викликаючи функцію “std::chrono::steady_clock::now()”, яка повертає позначку часу в поточний момент часу. Потім код викликає функцію “zapi::PackArchive”, щоб створити стислий архів директорії, визначеної параметром “folderPath”, і зберегти його у файлі, указаному параметром “archivePath”.

Після створення архіву код ініціалізує час завершення, знову викликаючи функцію “std::chrono::steady_clock::now()”. Потім код обчислює час, що минув між часом початку та закінчення, віднімаючи час початку від часу закінчення та перетворюючи результат у секунди. В кінці, код виводить час, що минув у мілісекундах, на консоль за допомогою “std::cout”.

Аналогічний код виконується в рядках 181-185.

```
165     if (argv[1][0] == 'p')
166     {
167         std::string folderPath = std::string(&argv[2][0]);
168         std::string archivePath = std::string(&argv[3][0]);
169 
170         auto start = std::chrono::steady_clock::now();
171         zapi::PackArchive pack(folderPath, archivePath);
172         auto end = std::chrono::steady_clock::now();
173         std::chrono::duration<double> elapsed_seconds = end-start;
174         std::cout << "Elapsed time: " << elapsed_seconds.count() * 1000.0 << " ms\n";
175     }
176     else if (argv[1][0] == 'x')
177     {
178         std::string folderPath = std::string(&argv[2][0]);
179         std::string archivePath = std::string(&argv[3][0]);
180 
181         auto start = std::chrono::steady_clock::now();
182         zapi::UnpackArchive unpack(folderPath, archivePath);
183         auto end = std::chrono::steady_clock::now();
184         std::chrono::duration<double> elapsed_seconds = end-start;
185         std::cout << "Elapsed time: " << elapsed_seconds.count() * 1000.0 << " ms\n";
186     }
```

Рисунок 3.3.1 — Код для фіксування часу роботи програми

Було створено архів з файлу “enwik8” (текстовий файл, що містить англійський текст і знаки пунктуації, який використовується як стандартний тест для алгоритмів стиснення даних. Він містить загалом 100 000 000 байт даних, включаючи пробіли, знаки пунктуації та комбінацію великих і малих літер.). Після чого було зафіксовано час створення архіву: *2154.46 мілісекунди*. Це доволі

небагато, у порівнянні, наприклад, з кодуванням файлу “enwik8” архіватором 7-Zip (*~3.5 секунди*), зображено на рисунку 3.3.2.

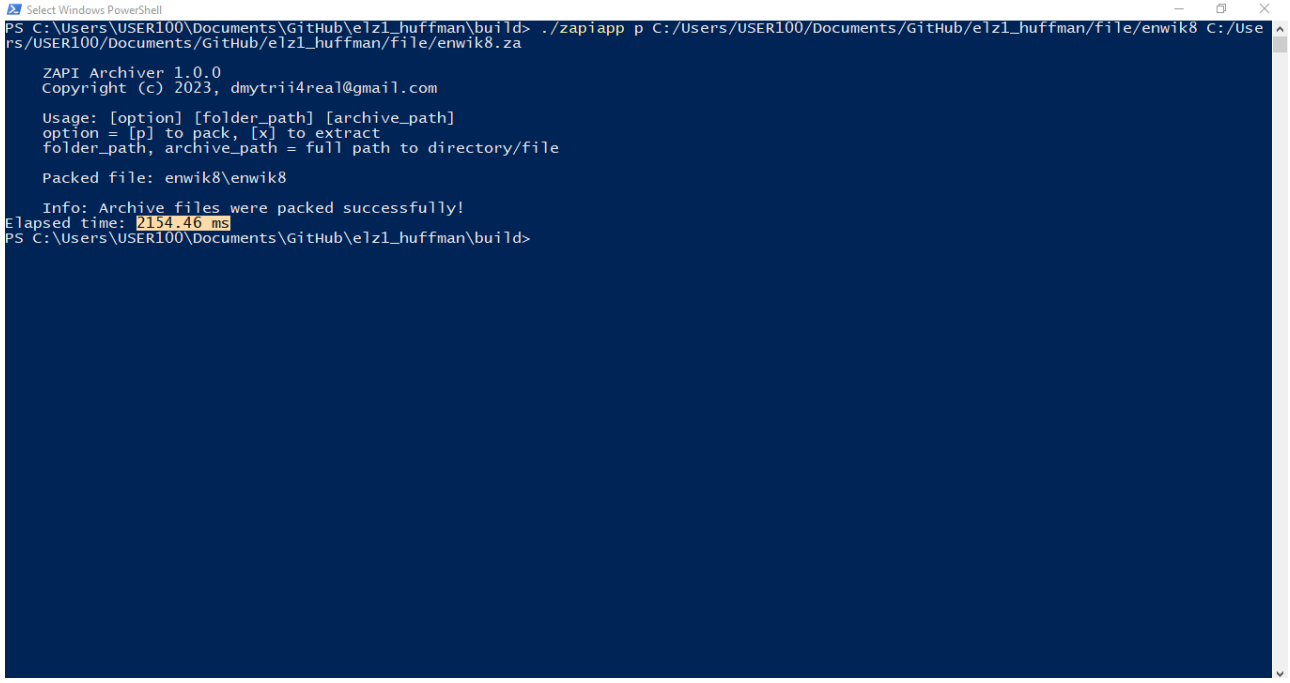


Рисунок 3.3.2 — Час створення архіву з файлу “enwik8”

Так само було зафіксовано час розпакування файлу “enwik8”. Результат склав: *2468.56 мілісекунди*. Цей доволі небагато, у порівнянні, наприклад, з декодуванням файлу “enwik8” архіватором 7-Zip (*~2.5 секунди*), зображено на рисунку 3.3.3.

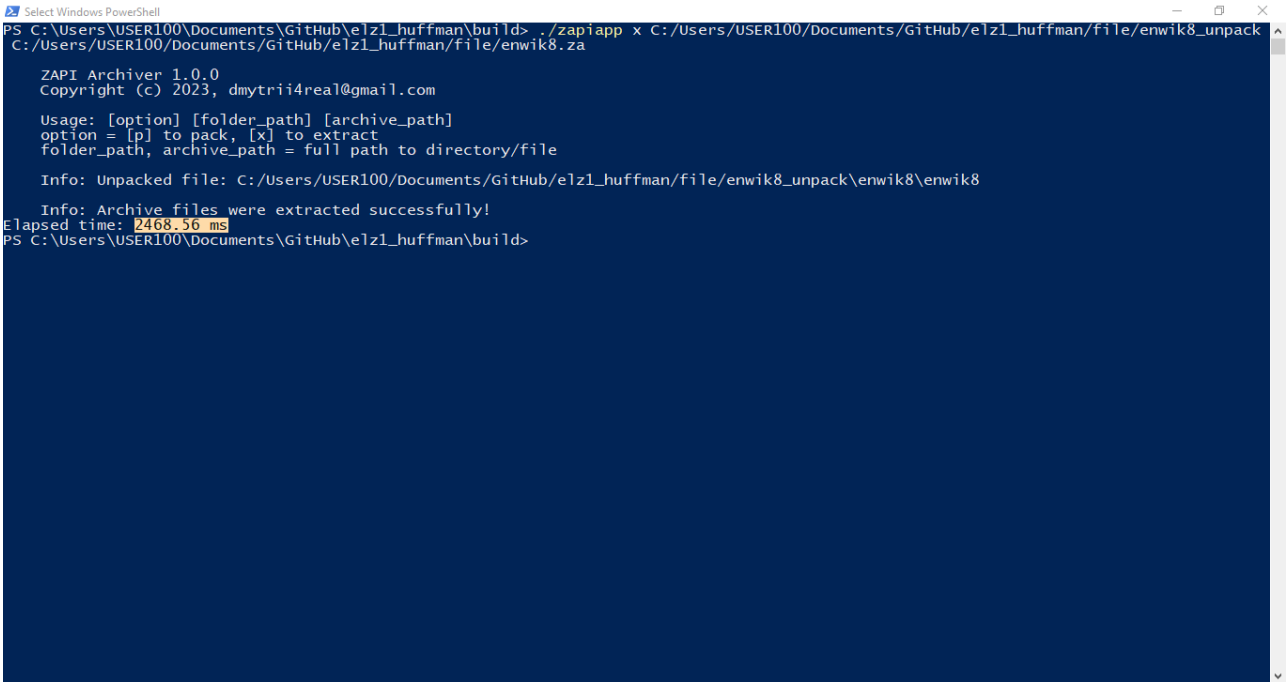


Рисунок 3.3.3 — Час розпакування файлу “enwik8”

Розмір архівованого файлу “enwik8” склав: *48 594 504 байт* (рисунок 3.3.4). Що є доволі непоганим результатом, у порівнянні, наприклад, з архіватором 7-Zip (розмір архівованого файлу склав *33 406 833 байт*).

Метою роботи, окрім збільшення швидкості архівування файлів також було покращення якості стиснення за рахунок ентропійного кодування. Результати порівняння вказані на рисунках: рисунок 3.3.4 використовує алгоритм LZ4 разом з методом ентропійного кодування Хаффмена, а рисунок 3.3.5 використовує лише алгоритм LZ4.

Отже, за рахунок ентропійного кодування Хаффмена, коефіцієнт стиснення було покращено в *1.18 разів*.

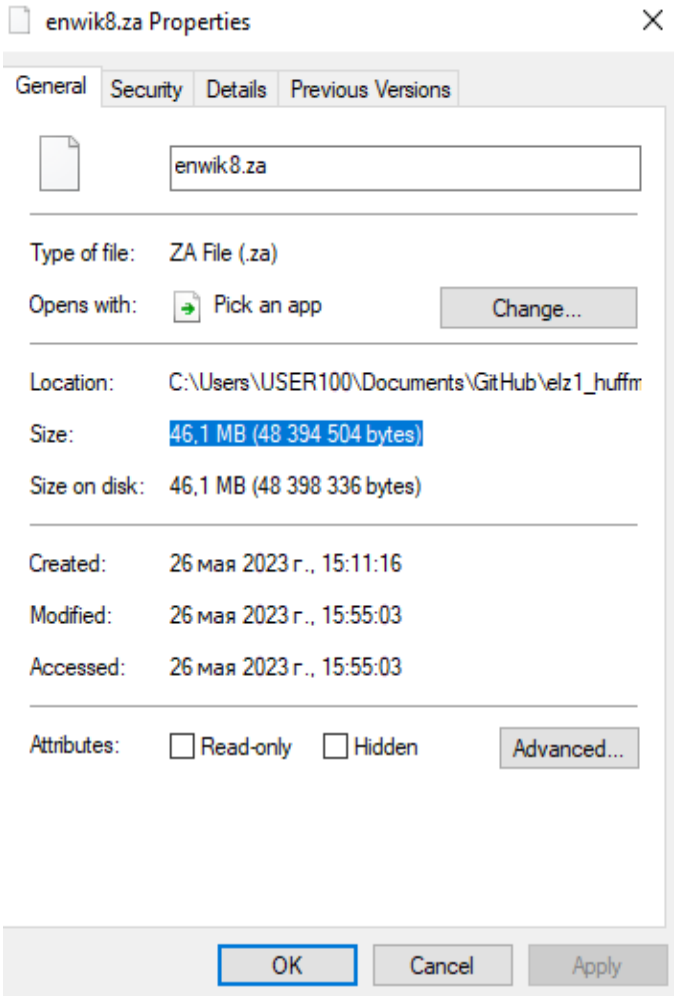


Рисунок 3.3.4 — Файл “enwik8” стиснутий алгоритмом LZ4 разом з методом ентропійним кодування Хаффмена

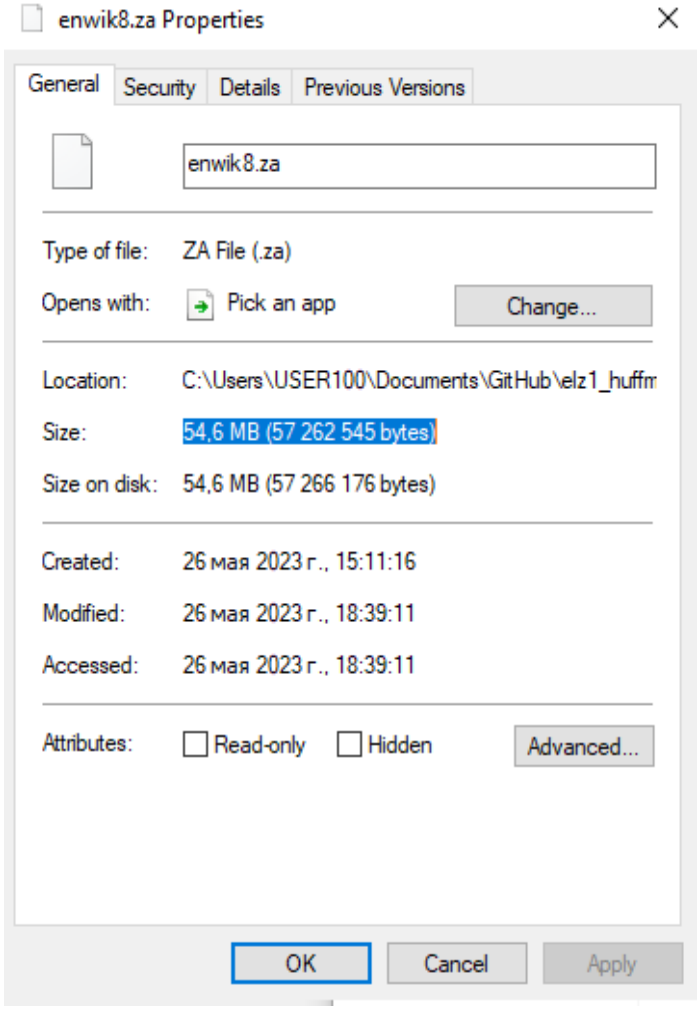


Рисунок 3.3.5 — Файл “enwik8” стиснутий лише алгоритмом LZ4

ВИСНОВКИ

Результатом дипломної роботи є програмне забезпечення для архівування даних. У роботі було покроково описано, яким чином можливо створити програмне забезпечення з архівації даних.

Був проведений аналіз існуючих засобів з архівування файлів і визначено основні вимоги до майбутнього програмного забезпечення.

Було розглянуто основні матеріали для розробки програми-архіватора. Проаналізовано та обрано для реалізації такі інструменти: мову програмування C++, компілятор MinGW, середу розробки Visual Studio Code та програму для збирання проектів CMake.

Було спроектовано програму-архіватор.

За результатами проектування було створено додаток, що відповідає поставленим вимогам.

Отже, результатом дипломної роботи є програмне забезпечення з архівування файлів, що пропонує швидкісне та якісне стиснення даних разом з архівацією файлів та відповідає поставленим вимогам. Додаток може бути покращений додатковим функціоналом та офіційно викладений.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Visual Studio Code [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/.
2. C++ Reference [Електронний ресурс] – Режим доступу до ресурсу: https://en.cppreference.com/w/.
3. MinGW-w64 Compiler [Електронний ресурс] – Режим доступу до ресурсу: https://www.mingw-w64.org/.
4. CMake [Електронний ресурс] – Режим доступу до ресурсу: https://cmake.org/.
5. Інформатика [Електронний ресурс] – Режим доступу до ресурсу: https://yevshan.com.ua/info/001/content/content2.html.
6. Системи числення [Електронний ресурс] – Режим доступу до ресурсу: https://sites.google.com/site/sistemicislennaveronika/dvijkova-sistema-cislenna.
7. LZ4 - Extremely fast compression [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/lz4/lz4.
8. Huffman Code [Електронний ресурс] – Режим доступу до ресурсу: https://brilliant.org/wiki/huffman-encoding/.
9. Дискретні джерела інформації [Електронний ресурс] – Режим доступу до ресурсу: https://studfile.net/preview/5465934/page:3/.
10. МІКРОПРОЦЕСОРНІ ТА МІКРОКОНТРОЛЕРНІ СИСТЕМИ [Електронний ресурс] – Режим доступу до ресурсу: https://ela.kpi.ua/bitstream/123456789/31216/1/MP_ta_MKS_1.pdf.
11. ТЕОРІЯ ІНФОРМАЦІЇ І КОДУВАННЯ: КУРС ЛЕКЦІЙ [Електронний ресурс] – Режим доступу до ресурсу: https://ela.kpi.ua/bitstream/123456789/41907/1/Kovalenko_AE_TIK_KursLecY20.pdf.
12. СИСТЕМИ ЗБОРУ ДАНИХ ТА ЇХ КОМПАКТНОГО ПРЕДСТАВЛЕННЯ [Електронний ресурс] – Режим доступу до ресурсу: https://er.chdtu.edu.ua/bitstream/ChSTU/3515/1/%D0%9A%D0%BE%D0%BD

%D1%81%D0%BF%D0%B5%D0%BA%D1%82%20%D0%BB%D0%B5%D0%BA%D1%86%D0%B8%D0%B9.pdf.

ДОДАТОК А

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для архівації даних на основі алгоритму сімейства LZ77 та ентропійного кодувальника мовою C++

Виконав студент 5 курсу
групи ППЗ-51
Сіраченко Дмитрій Владиславович
Керівник роботи
К.е.н, доц, доцент кафедри ІПЗ Аверічев Ігор Миколайович

Київ – 2023

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи:** підвищення швидкості архівування файлів за рахунок алгоритму LZ4 та якості стиснення за рахунок ентропійного кодування.
- **Об’єкт дослідження:** процес архівування файлів файлової системи.
- **Предмет дослідження:** додаток, що використовує швидкісний алгоритм LZ4 з ентропійним кодуванням для архівування файлів.

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проаналізувати теоретичні дослідження в сфері архівації даних.
2. Визначити переваги та можливості сучасних програмних засобів.
3. Дослідити алгоритми стиснення даних, які використовуються в програмах-архіваторах, та визначити найшвидкісніший з них.
4. Спроектувати та розробити застосунок для архівації даних.

АНАЛІЗ ІСНУЮЧИХ ЗАСОБІВ

	7-Zip	**WinZip**	**ZAPI Archiver**
Консольний додаток	ні	ні	так
Портативність (не треба інсталювати додаток)	так	ні	так
Компактність (розмір програми менший за 15 МіБ)	ні	ні	так
Час витрачений на кодування (файл ‘enwik8’)	~3.5 с	~4 с	~2.5 с

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Створення та розпаковування файлових архівів.
2. Швидкісне (швидше за 7-Zip та WinZip) кодування даних в файлових архівах.
3. Використання ентропійного кодування в файлових архівах.

1. Нефункціональні вимоги:
2. Використання мови C++ та вбудованих в неї бібліотек.
3. Використання алгоритму LZ4 для швидкісного стиснення даних.
4. Використання методу Хаффмена для ентропійного кодування.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Компілятор

Програма для автоматизації збирання проекту

АНАЛІЗ ФАЙЛІВ ТА КЛАСІВ

samples/main.cpp — файл, що містить функцію точки входу в консольний додаток програми-архіватора та використовує основні класи програми-архіватора для обробки файлів, що введені в якості консольних аргументів.

codecs/huffman.hpp — файл, що описує інтерфейс кодування та декодування за методом Хаффмена.

codecs/huffman.cpp — файл, що реалізує інтерфейс кодування та декодування за методом Хаффмена.

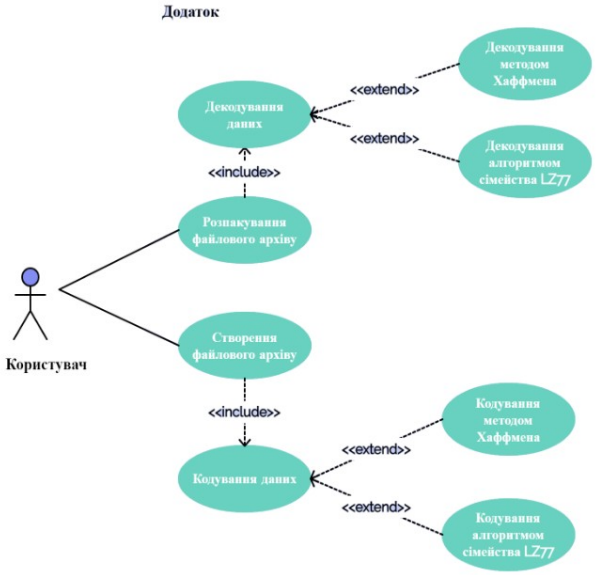
codecs/lz4.h — файл, що описує інтерфейс кодування та декодування за алгоритмом LZ4.

codecs/lz4.c — файл, що реалізує інтерфейс кодування та декодування за алгоритмом LZ4.

source/zapi.hpp — файл, що містить основні класи програми-архіватора (Buffer, PackArchive, UnpackArchive, Codec_Interface, CodecLZ77WithEntropy).

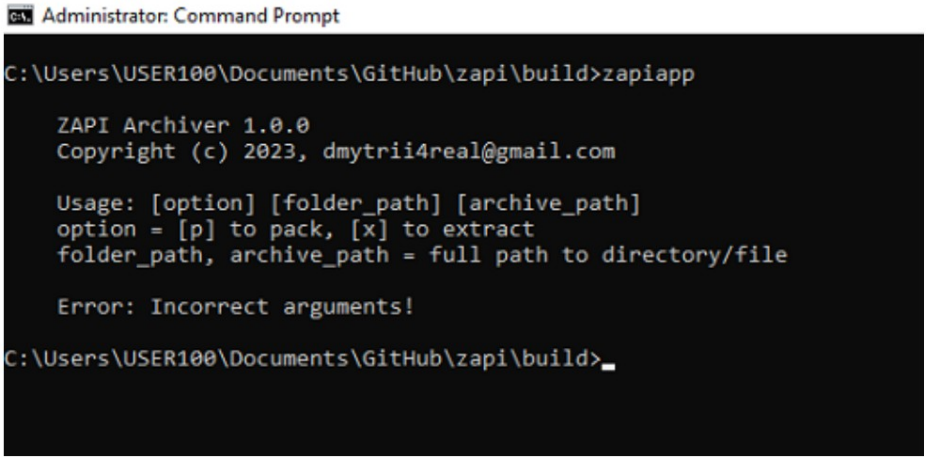
source/zapi.cpp — файл, що містить реалізацію основних класів програми-архіватора.

ДІАГРАМА ПРЕЦЕДЕНТІВ



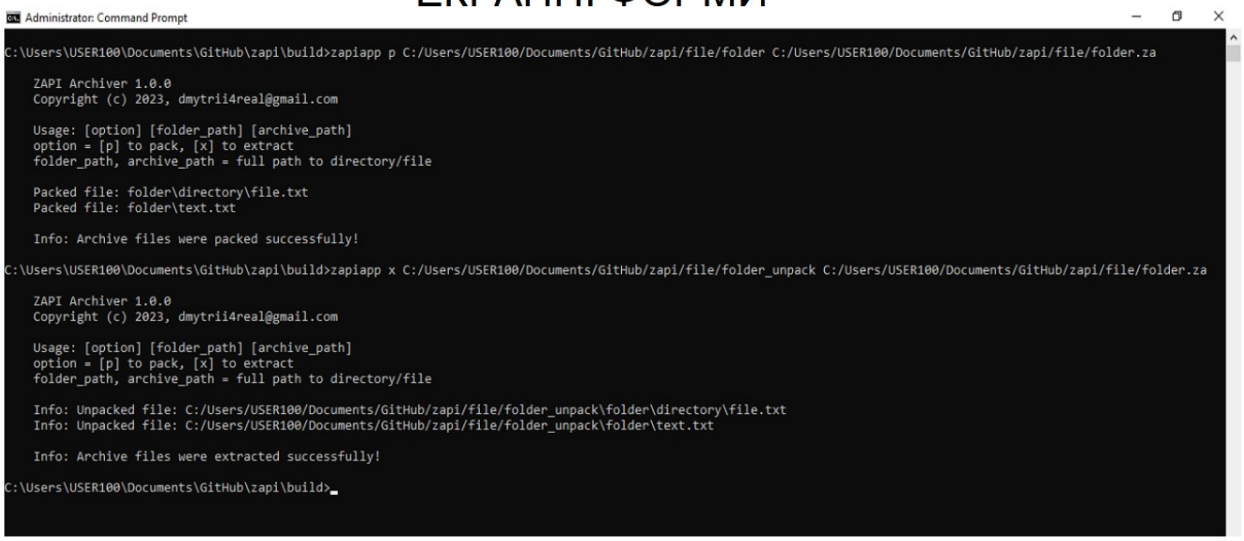
ДІАГРАМА КЛАСІВ

ЕКРАННІ ФОРМИ



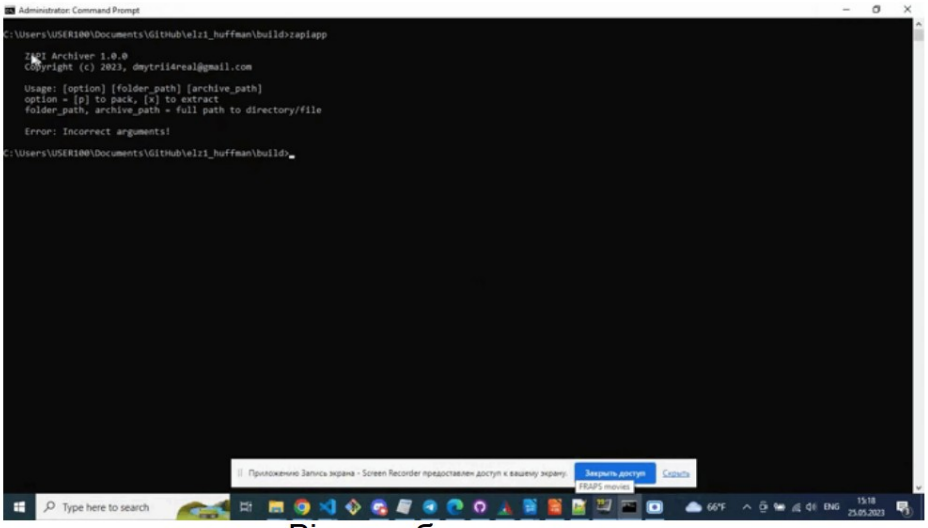
Вивід інструкцій до використання консольного додатку

ЕКРАННІ ФОРМИ



Команди пакування та розпакування папок файлової системи

ЕКРАННІ ФОРМИ



Відео роботи програми

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1 Аверічев І.М., Сіраченко Д.В. Розробка програмного забезпечення для архівації даних на основі алгоритму сімейства LZ77 // Всеукраїнська науково-технічна конференція “Застосування програмного забезпечення в інфокомунікаційних технологіях” - Київ: ДУТ, 2023 - с.;

2 Аверічев І.М., Сіраченко Д.В. Розробка програмного забезпечення для архівації даних за допомогою ентропійного кодування // Всеукраїнська науково-технічна конференція “Застосування програмного забезпечення в інфокомунікаційних технологіях” - Київ: ДУТ, 2023 - с.

ВИСНОВКИ

1. Проаналізовано теоретичні дослідження в сфері архівації даних.
2. Визначено переваги та можливості сучасних програмних засобів.
3. Досліджено алгоритми стиснення даних, які використовуються в програмах-архіваторах, та визначено найшвидкісніший з них.
4. Спроектовано та розроблено застосунок для архівації даних.

ДЯКУЮ ЗА УВАГУ!

ДОДАТОК Б

samples/main.cpp:

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <string>
#include <cstring>
#include <windows.h>
#include "../codecs/elz1.h"
#include "../codecs/huffman.hpp"
#include "../source/zapi.hpp"

int main(int argc, char** argv)
{
  std::cout << std::endl;
  std::cout << "    ZAPI Archiver 1.0.0" << std::endl;
  std::cout << "    Copyright (c) 2023, dmytrii4real@gmail.com" << std::endl;
  std::cout << std::endl;
  std::cout << "    Usage: [option] [folder_path] [archive_path]" << std::endl;
  std::cout << "    option = [p] to pack, [x] to extract" << std::endl;
  std::cout << "    folder_path, archive_path = full path to directory/file" <<
std::endl;
  std::cout << std::endl;

  if (argc < 4 || (argv[1][0] != 'p' && argv[1][0] != 'x'))
  {
    std::cout << "    Error: Incorrect arguments!" << std::endl;

    return 0;
  }

  if (argv[1][0] == 'p')
  {
    std::string folderPath = std::string(&argv[2][0]);
    std::string archivePath = std::string(&argv[3][0]);

    auto start = std::chrono::steady_clock::now();
    zapi::PackArchive pack(folderPath, archivePath);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed_seconds = end-start;
    std::cout << "    Elapsed time: " << elapsed_seconds.count() * 1000.0 << " ms\n";
  }
  else if (argv[1][0] == 'x')
  {
    std::string folderPath = std::string(&argv[2][0]);
    std::string archivePath = std::string(&argv[3][0]);

    auto start = std::chrono::steady_clock::now();
```

```
    zapi::UnpackArchive unpack(folderPath, archivePath);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed_seconds = end-start;
    std::cout << "    Elapsed time: " << elapsed_seconds.count() * 1000.0 << " ms\n";
  }

  return 0;
}
```

codecs/huffman.hpp:

```
#include <cstdint>
#include <cstring>

struct Symbol
{
  unsigned Char;
  unsigned Freq;
  unsigned Skip;
  unsigned Left;
  unsigned Right;
  unsigned BitLength;
  unsigned Code;
  unsigned Weight;
};

void HuffmanAssignCode(Symbol* tree, unsigned index, uint64_t code, unsigned
codeBitSize);
void HuffmanEncodeData(uint8_t* inputData, uint8_t* outputData, size_t
inputDataByteSize, size_t& outputDataByteSize);
void HuffmanDecodeData(uint8_t* inputData, uint8_t* outputData, size_t
inputDataByteSize, size_t& outputDataByteSize);
```

codecs/huffman.cpp:

```
#include "huffman.hpp"

void HuffmanAssignCode(Symbol* tree, unsigned index, uint64_t code, unsigned
codeBitSize)
{
  if (tree[index].Char == 256)
  {
    HuffmanAssignCode(tree, tree[index].Left, code, codeBitSize + 1);
    HuffmanAssignCode(tree, tree[index].Right, code | (1 << codeBitSize), codeBitSize
+ 1);
  }
  else
  {
```

```
    tree[index].Code = code;
    tree[index].BitLength = codeBitSize;
  }
}

void HuffmanEncodeData(uint8_t* inputData, uint8_t* outputData, size_t
inputDataByteSize, size_t& outputDataByteSize)
{
  uint8_t* output = outputData;

  Symbol symbols[512] = {};

  // Count symbol frequencies
  for (int i = 0; i < inputDataByteSize; i++)
  {
    symbols[inputData[i]].Char = inputData[i];
    symbols[inputData[i]].Freq += 1;
  }

  // Store symbols only with non-zero frequencies
  unsigned symbolCount = 0;
  for (int i = 0; i < 256; i++) {
    if (symbols[i].Freq != 0) {
      symbols[symbolCount++] = symbols[i];
    }
  }

  // Radix sort symbol frequencies
  Symbol sortedSymbols[256] = {};
  for (int i = 0; i < 32; i++)
  {
    unsigned bit0Count = 0;
    unsigned bit1Count = 0;
    unsigned bits0[256] = {};
    unsigned bits1[256] = {};

    for (int j = 0; j < symbolCount; j++)
    {
      if (((symbols[j].Freq >> i) & 1) == 1) {
        bits1[bit1Count++] = j;
      } else {
        bits0[bit0Count++] = j;
      }
    }

    unsigned cnt = 0;
    for (int j = 0; j < bit0Count; j++, cnt++) {
      sortedSymbols[cnt] = symbols[bits0[j]];
    }
    for (int j = 0; j < bit1Count; j++, cnt++) {
      sortedSymbols[cnt] = symbols[bits1[j]];
```

```
  }

  memcpy(&symbols[0], &sortedSymbols[0], symbolCount * sizeof(Symbol));
}

// Construct tree
unsigned totalNodeCount = symbolCount - 1;
unsigned curNodeCount = symbolCount;
for (int i = 0; i < totalNodeCount; i++)
{
  unsigned minIndex0 = 0;
  unsigned minIndex1 = 0;
  unsigned freq0 = 0;

  for (int j = 0; j < curNodeCount; j++)
  {
    if (symbols[j].Freq < symbols[minIndex0].Freq) {
      minIndex0 = j;
    }
  }

  freq0 = symbols[minIndex0].Freq;
  symbols[minIndex0].Freq = 0xffffffff;

  for (int j = 0; j < curNodeCount; j++)
  {
    if (symbols[j].Freq < symbols[minIndex1].Freq) {
      minIndex1 = j;
    }
  }

  // Add new node
  symbols[curNodeCount].Char = 256;
  symbols[curNodeCount].Freq = freq0 + symbols[minIndex1].Freq;
  symbols[curNodeCount].Left = minIndex0;
  symbols[curNodeCount].Right = minIndex1;
  symbols[minIndex0].Weight = freq0;
  symbols[minIndex1].Weight = symbols[minIndex1].Freq;
  symbols[minIndex1].Freq = 0xffffffff;
  curNodeCount += 1;
}

HuffmanAssignCode(&symbols[0], curNodeCount - 1, 0, 0);

// Build code table
struct Code
{
  unsigned Bits;
  unsigned Length;
};
```

```
  Code table[256] = {};
  for (int i = 0; i < symbolCount; i++)
  {
    table[symbols[i].Char].Bits = symbols[i].Code;
    table[symbols[i].Char].Length = symbols[i].BitLength;
  }

  // Write Huffman tree to output
  unsigned inputByteSize = (unsigned)inputDataByteSize;
  ((unsigned*)output)[0] = inputByteSize;
  ((unsigned*)output)[1] = curNodeCount;
  output += 8;
  memcpy(&output[0], symbols, curNodeCount * sizeof(Symbol));
  output += curNodeCount * sizeof(Symbol);

  // Encode input data
  unsigned long long bitBufferBitSize = 0;
  unsigned long long bitBuffer = 0;
  for (int i = 0; i < inputDataByteSize; i++)
  {
    bitBuffer |= (unsigned long long)table[inputData[i]].Bits << bitBufferBitSize;
    bitBufferBitSize += table[inputData[i]].Length;

    while (bitBufferBitSize >= 32)
    {
      ((unsigned*)output)[0] = bitBuffer & 0xffffffff;
      output += 4;
      bitBuffer >>= 32ull;
      bitBufferBitSize -= 32ull;
    }
  }

  while (bitBufferBitSize > 0)
  {
    ((unsigned*)output)[0] = bitBuffer & 0xffffffff;
    output += 4;
    bitBuffer >>= 32ull;

    if (bitBufferBitSize > 32ull) {
      bitBufferBitSize -= 32ull;
    } else {
      break;
    }
  }

  outputDataByteSize = output - outputData;
}

void HuffmanDecodeData(uint8_t* inputData, uint8_t* outputData, size_t
inputDataByteSize, size_t& outputDataByteSize)
{
```

```
  uint8_t* input = (uint8_t*)inputData;
  uint8_t* output = (uint8_t*)outputData;

  Symbol tree[512] = {};
  unsigned symbolCount = ((unsigned*)input)[0];
  unsigned nodeCount = ((unsigned*)input)[1];
  input += 8;

  memcpy(&tree[0], &input[0], nodeCount * sizeof(Symbol));
  input += nodeCount * sizeof(Symbol);

  unsigned long long bitBufferBitSize = 0;
  unsigned long long bitBuffer = 0;

  uint32_t* inputDWord = (uint32_t*)input;
  uint32_t* inputDWordBoundary = (uint32_t*)(input + inputDataByteSize);
  uint8_t* outputLast = output + symbolCount;

  while (output < outputLast)
  {
    unsigned curIndex = nodeCount - 1;
    unsigned symbol = 0;

    do
    {
      while (bitBufferBitSize < 32 && inputDWord < inputDWordBoundary)
      {
        bitBuffer |= (unsigned long long)(*inputDWord++) << bitBufferBitSize;
        bitBufferBitSize += 32ull;
      }

      unsigned bit = bitBuffer & 1ull;
      bitBuffer >>= 1ull;
      bitBufferBitSize -= 1ull;

      if (bit == 0) {
        curIndex = tree[curIndex].Left;
      } else {
        curIndex = tree[curIndex].Right;
      }

      symbol = tree[curIndex].Char;
    } while (symbol == 256);

    *output++ = symbol;
  }

  outputDataByteSize = output - outputData;
}
```

codecs/lz4.h:
https://github.com/lz4/lz4/blob/dev/lib/lz4.h

codecs/lz4.h:
https://github.com/lz4/lz4/blob/dev/lib/lz4.c

source/zapi.hpp:

```
#include <cstdint>
#include <string>

namespace zapi
{
  class Buffer
  {
    public:
      explicit Buffer(size_t byteSize, void* data);
      ~Buffer();

      size_t& GetSizeRef() { return _byteSize; }
      void* GetData() { return _data; }

    private:
      size_t _byteSize;
      void* _data;
  };

  class PackArchive
  {
    public:
      explicit PackArchive(std::string folderPath, std::string archivePath);
      ~PackArchive();

      std::string& GetFolderPathRef() { return _folderPath; }
      std::string& GetArchivePathRef() { return _archivePath; }

    private:
      std::string _folderPath;
      std::string _archivePath;
  };

  class UnpackArchive
  {
    public:
      explicit UnpackArchive(std::string folderPath, std::string archivePath);
      ~UnpackArchive();

      std::string& GetFolderPathRef() { return _folderPath; }
      std::string& GetArchivePathRef() { return _archivePath; }
```

```
    private:
      std::string _folderPath;
      std::string _archivePath;
  };

  class Codec_Interface
  {
    public:
      Codec_Interface() {}
      ~Codec_Interface() {}

      virtual void EncodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer) = 0;
      virtual void DecodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer) = 0;
  };

  class CodecLZ77WithEntropy : public Codec_Interface
  {
    public:
      CodecLZ77WithEntropy() {}
      ~CodecLZ77WithEntropy() {}

      void EncodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer) override final;
      void DecodeBuffer(Buffer* inputBuffer, Buffer* outputBuffer) override final;
  };
}
```

source/zapi.cpp:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <windows.h>
#include <cstdlib>
#include "zapi.hpp"
#include "../codecs/elz1.h"
#include "../codecs/lz4.h"
#include "../codecs/huffman.hpp"

void PackFile(std::ofstream& archiveFileStream, const char* filePath, const char*
localFilePath)
{
  std::ifstream fileStream(filePath, std::ios::in | std::ios::binary);
  fileStream.seekg(0, std::ios::end);
  size_t fileByteSize = fileStream.tellg();
  fileStream.seekg(0, std::ios::beg);
  void* fileMemory = malloc(fileByteSize);
  fileStream.read((char*)fileMemory, fileByteSize);
  fileStream.close();
```

```
  zapi::Buffer bInput(fileByteSize, fileMemory);
  zapi::Buffer bOutput(100 * 1024 * 1024, nullptr);
  zapi::CodecLZ77WithEntropy compress;
  compress.EncodeBuffer(&bInput, &bOutput);

  char fileLocalPathStr[256] = {};
  memcpy(&fileLocalPathStr[0], &localFilePath[0], strlen(localFilePath));

  archiveFileStream.write((const char*)&bOutput.GetSizeRef(), sizeof(unsigned));
  archiveFileStream.write((const char*)&fileLocalPathStr[0], 256);
  archiveFileStream.write((const char*)bOutput.GetData(), bOutput.GetSizeRef());

  free(fileMemory);
}

void UnpackFile(std::ifstream& archiveFileStream, const char* folderPath)
{
  std::string filePath(folderPath); unsigned fileByteSize = 0; char
fileLocalPathStr[256] = {}; void* fileMemory = nullptr;

  archiveFileStream.read((char*)&fileByteSize, sizeof(unsigned));
  archiveFileStream.read((char*)&fileLocalPathStr[0], 256);
  fileMemory = malloc(fileByteSize);
  archiveFileStream.read((char*)fileMemory, fileByteSize);

  filePath = filePath + '\\' + fileLocalPathStr;
  std::string localFolderName;
  char appendStr[2] = { filePath[0], 0 };
  localFolderName.append(&appendStr[0]);

  for (int i = 1; i < filePath.size();) {
    char appendStr[2] = { filePath[i], 0 };
    localFolderName.append(&appendStr[0]);
    if (filePath[i - 1] == '/' || filePath[i - 1] == '\\') {
      i += 1;
      while (filePath[i] != '/' && filePath[i] != '\\' && i < filePath.size()) {
        char appendStr[2] = { filePath[i], 0 };
        localFolderName.append(&appendStr[0]);
        i += 1;
      }
      if (i != filePath.size()) { CreateDirectoryA(localFolderName.c_str(), NULL); }
else { break; }
    } else {
      i += 1;
    }
  }

  // Decompress file data
  zapi::Buffer bInput(fileByteSize, fileMemory);
  zapi::Buffer bOutput(100 * 1024 * 1024, nullptr);
```

```
  zapi::CodecLZ77WithEntropy decompress;
  decompress.DecodeBuffer(&bInput, &bOutput);

  std::ofstream fileStream(filePath.c_str(), std::ios::out | std::ios::binary);
  fileStream.write((const char*)bOutput.GetData(), bOutput.GetSizeRef());
  fileStream.close();

  free(fileMemory);

  std::cout << "    Info: Unpacked file: " << filePath << std::endl;
}

void ReadDirectories(std::ofstream& archiveFileStream, const std::string& path, const
std::string& localPath)
{
  WIN32_FIND_DATAA findData;
  HANDLE hFind = FindFirstFileA((path + "\\*").c_str(), &findData);

  if (hFind != INVALID_HANDLE_VALUE) {
    do {
      if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
        if (strcmp(findData.cFileName, ".") != 0 && strcmp(findData.cFileName,
"..") != 0) {
          std::string directoryPath = path + "\\" + findData.cFileName;
          std::string newLocalPath = localPath + "\\" + findData.cFileName;
          ReadDirectories(archiveFileStream, directoryPath, newLocalPath);
        }
      } else {
        static unsigned fileCount = 0; fileCount += 1;

        std::string newLocalPath = localPath + "\\" + findData.cFileName;

        auto streamPos = archiveFileStream.tellp();
        archiveFileStream.seekp(0, std::ios::beg);
        archiveFileStream.write((const char*)&fileCount, sizeof(unsigned));
        archiveFileStream.seekp(streamPos);

        PackFile(archiveFileStream, std::string(path + "\\" +
findData.cFileName).c_str(), newLocalPath.c_str());

        std::cout << "    Packed file: " << newLocalPath << std::endl;
      }
    } while (FindNextFileA(hFind, &findData));
    FindClose(hFind);
  }
}

zapi::Buffer::Buffer(size_t byteSize, void* data)
{
  if (byteSize != 0)
  {
```

```
    _byteSize = byteSize;
    _data = malloc(byteSize);

    if (data != nullptr) {
      memcpy(_data, data, byteSize);
    }
  }
  else
  {
    _byteSize = 0;
    _data = nullptr;
  }
}

zapi::Buffer::~Buffer()
{
  if (_data != nullptr) {
    free(_data);
  }
}

zapi::PackArchive::PackArchive(std::string folderPath, std::string archivePath)
{
  _folderPath = folderPath, _archivePath = archivePath;
  std::string folderName, reversedFolderName;

  int i = _folderPath.size() - 1;
  while (_folderPath[i] != '/') {
    char appendStr[2] = { _folderPath[i], 0 };
    reversedFolderName.append(&appendStr[0]);
    i -= 1;
  }

  i = reversedFolderName.size() - 1;
  folderName.reserve(reversedFolderName.size());
  while (i > -1) {
    char appendStr[2] = { reversedFolderName[i], 0 };
    folderName.append(&appendStr[0]);
    i -= 1;
  }

  unsigned fileCount = 0;
  std::ofstream archiveFileStream(_archivePath, std::ios::out | std::ios::binary);
  if (archiveFileStream.fail()) { std::cout << "    Error: Archive file not found!"
<< std::endl; return; }

  archiveFileStream.write((const char*)&fileCount, sizeof(unsigned));
  ReadDirectories(archiveFileStream, _folderPath, folderName);
  archiveFileStream.close();
```

```
  std::cout << std::endl << "    Info: Archive files were packed successfully!" <<
std::endl;
}

zapi::PackArchive::~PackArchive()
{
}

zapi::UnpackArchive::UnpackArchive(std::string folderPath, std::string archivePath)
{
  _folderPath = folderPath;
  _archivePath = archivePath;

  unsigned fileCount = 0;
  std::ifstream archiveFileStream(_archivePath, std::ios::in | std::ios::binary);
  if (archiveFileStream.fail()) {
    std::cout << "    Error: Archive file not found!" << std::endl;

    return;
  }

  archiveFileStream.read((char*)&fileCount, sizeof(unsigned));
  while (fileCount > 0)
  {
    UnpackFile(archiveFileStream, _folderPath.c_str());
    fileCount -= 1;
  }
  archiveFileStream.close();

  std::cout << std::endl << "    Info: Archive files were extracted successfully!" <<
std::endl;
}

zapi::UnpackArchive::~UnpackArchive()
{
}

void zapi::CodecLZ77WithEntropy::EncodeBuffer(Buffer* inputBuffer, Buffer*
outputBuffer)
{
  {
    // LZ4 coding
    size_t lz4TempBufferByteSize = 128 * 1024 * 1024;
    Buffer lz4TempBuffer(lz4TempBufferByteSize, nullptr);
    size_t lz4CodedByteSize = LZ4_compress_default((const char*)inputBuffer-
>GetData(),
      (char*)lz4TempBuffer.GetData(), inputBuffer->GetSizeRef(), inputBuffer-
>GetSizeRef());
    lz4TempBuffer.GetSizeRef() = lz4CodedByteSize;

    {
```

```
      // Huffman coding
      size_t huffmanTempBufferByteSize = 128 * 1024 * 1024;
      Buffer huffmanTempBuffer(huffmanTempBufferByteSize, nullptr);
      HuffmanEncodeData((uint8_t*)lz4TempBuffer.GetData(),
        (uint8_t*)huffmanTempBuffer.GetData(), lz4TempBuffer.GetSizeRef(),
huffmanTempBufferByteSize);
      huffmanTempBuffer.GetSizeRef() = huffmanTempBufferByteSize;

      // Copy data to output buffer
      memcpy(outputBuffer->GetData(), huffmanTempBuffer.GetData(),
huffmanTempBuffer.GetSizeRef());
      outputBuffer->GetSizeRef() = huffmanTempBuffer.GetSizeRef();
    }
  }
}

void zapi::CodecLZ77WithEntropy::DecodeBuffer(Buffer* inputBuffer, Buffer*
outputBuffer)
{
  {
    // Huffman decoding
    size_t huffmanTempBufferByteSize = 128 * 1024 * 1024;
    Buffer huffmanTempBuffer(huffmanTempBufferByteSize, nullptr);
    HuffmanDecodeData((uint8_t*)inputBuffer->GetData(),
      (uint8_t*)huffmanTempBuffer.GetData(), inputBuffer->GetSizeRef(),
huffmanTempBufferByteSize);
    huffmanTempBuffer.GetSizeRef() = huffmanTempBufferByteSize;

    {
      // LZ4 decoding
      size_t lz4TempBufferByteSize = 128 * 1024 * 1024;
      Buffer lz4TempBuffer(lz4TempBufferByteSize, nullptr);
      size_t lz4DecodedByteSize = LZ4_decompress_safe((const
char*)huffmanTempBuffer.GetData(),
        (char*)lz4TempBuffer.GetData(), huffmanTempBuffer.GetSizeRef(),
lz4TempBufferByteSize);
      lz4TempBuffer.GetSizeRef() = lz4DecodedByteSize;

      // Copy data to output buffer
      memcpy(outputBuffer->GetData(), lz4TempBuffer.GetData(), lz4DecodedByteSize);
      outputBuffer->GetSizeRef() = lz4DecodedByteSize;
    }
  }
}
```