**Отчёт по лабораторной работе №1**

**Сжатие данные**

Задание:

Написать программу, сжимающую данные из некоторого файла по алгоритму Хаффмана.

Программа должна выполнять следующие действия:

Сжатие данных

1. Открывать файл, подлежащий сжатию, как бинарный файл байтов.

2. Подсчитывать частоту вхождения каждого байта в файле.

3. По имеющимся частотам строить дерево Хаффмана.

4. Создавать новый файл-архив.

5. Записывать в файл-архив заголовок – байты с частотами их вхождения.

6. Последовательно считывать байты исходного файла, кодировать их и записывать в файл-архив.

Распаковка данных

1. Открывать файл-архив.

2. Считывать из файла-архива заголовок и строить по нему дерево Хаффмана.

3. Последовательно считывать байты файла-архива, анализировать байты побитно с помощью дерева Хаффмана и записывать найденные в дереве байты в новый файл.

Код программы:

|  |
| --- |
| GUI.h |
| private:  bool running = false;  String^ exists\_name = "";  /\*  \* Функция, преобразующая System::String^ в std::string  \*/  std::string to\_string(System::String^ ms\_string) {  return msclr::interop::marshal\_as<std::string>(ms\_string);  }  /\*  \* Метод-обёртка для выполнения сжатия в отдельном потоке  \*/  System::Void CompressTask() {  if (exists\_name != "")  archiver->compress(to\_string(filenameBox->Text), to\_string(exists\_name));  else  archiver->compress(to\_string(filenameBox->Text), to\_string(filenameBox->Text + ".gzp"));  running = false;  }  /\*  \* Метод-обёртка для выполнения разжатия в отдельном потоке  \*/  System::Void DecompressTask() {  if (exists\_name != "")  archiver->decompress(to\_string(filenameBox->Text), to\_string(exists\_name));  else  archiver->decompress(to\_string(filenameBox->Text), to\_string(filenameBox->Text->Replace(".gzp", "")));  running = false;  }  /\*  \* Метод, обновляющий прогрессбар  \*/  System::Void UpdateProgress(System::Object^ sender, System::EventArgs^ e) {  progressBar1->Value = archiver->getProgress();  if (progressBar1->Value == 0 && running) progressBar1->Style = ProgressBarStyle::Marquee;  else progressBar1->Style = ProgressBarStyle::Blocks;  }  System::Void buttonSelect\_Click(System::Object^ sender, System::EventArgs^ e) {  openFileDialog1->FileName = "";  openFileDialog1->ShowDialog();  filenameBox->Text = openFileDialog1->FileName;  String^ ext = filenameBox->Text->Split('.')[filenameBox->Text->Split('.')->Length - 1];  }    System::Void buttonComp\_Click(System::Object^ sender, System::EventArgs^ e) {  if (filenameBox->Text != "" && !running) {  if (IO::File::Exists(filenameBox->Text + ".gzp")) {  MessageBox::Show("Архив с таким именем уже существует! Выберите новый");  saveFileDialog1->ShowDialog();  exists\_name = saveFileDialog1->FileName;  }  else exists\_name = "";  running = true;  Task^ task = Task::Run(gcnew Action(this, &GUI::CompressTask));  }  }  System::Void buttonUnComp\_Click(System::Object^ sender, System::EventArgs^ e) {  if (filenameBox->Text != "" && !running) {  if (IO::File::Exists(filenameBox->Text->Replace(".gzp", ""))) {  MessageBox::Show("Файл с таким именем уже существует! Выберите новый");  saveFileDialog1->ShowDialog();  exists\_name = saveFileDialog1->FileName;  }  else exists\_name = "";  running = true;  Task^ task = Task::Run(gcnew Action(this, &GUI::DecompressTask));  }  }  };  } |
| archiver.cpp |
| #pragma once  #include <string>  #include <fstream>  #include <array>  #include "huffman.h"  #include "archiver.h"  void Archiver::updateProgress() {  progress = ((float)processed / (float)fsize) \* 100;  }  int Archiver::getProgress() {  return progress;  }  /\*  \* Метод сжатия файла  \*/  void Archiver::compress(std::string fileName, std::string outName) {  HuffmanTree hf\_tree;  std::ifstream file(fileName, std::ios::binary);  if (file.is\_open()) {  // Подсчёт размера файла для вычисления прогресса  file.seekg(0, std::ios::end);  fsize = file.tellg();  file.seekg(0, std::ios::beg);  // Создание дерева Хаффмана из байт  unsigned char byte;  while (file.read((char\*)(&byte), sizeof(byte)))  hf\_tree.addByte(byte);  hf\_tree.makeTree();  file.close();  std::ofstream out(outName, std::ios::binary);  // Запись типа файла  out.write(new char[4] {'G', 'N', 'E', 'G'}, 4);  std::array<int, BYTE\_MAX> freq = hf\_tree.getFreq();  for (int i = 0; i < freq.size(); i++)  out.write((char\*)(&freq[i]), sizeof(freq[i]));  // Запись сжатых данных  file.open(fileName, std::ios::binary);  unsigned char compByte = 0;  std::string buffer = "";  while (file.read((char\*)(&byte), sizeof(byte))) {  buffer += hf\_tree.getCode(byte);  while (buffer.size() >= 8) {  compByte = 0;  for (int i = 0; i < 8; ++i)  if (buffer[i] == '1')  compByte += (unsigned char)(pow(2, 7 - i));  out.write((char\*)(&compByte), sizeof(compByte));  buffer = buffer.substr(8);  }  processed++;  updateProgress();  }  compByte = 0;  for (int i = 0; i < buffer.size(); ++i)  if (buffer[i] == '1')  compByte += (unsigned char)(pow(2, 7 - i));  out.write((char\*)(&compByte), sizeof(compByte));  file.close();  out.close();  }  // Сброс прогресса по завершению  processed = 0;  updateProgress();  }  /\*  \* Метод разжатия файла  \*/  void Archiver::decompress(std::string fileName, std::string outName) {  HuffmanTree hf\_tree;  std::ifstream file(fileName, std::ios::binary);  if (file.is\_open()) {  unsigned char byte;  // Подсчёт размера файла для вычисления прогресса  file.seekg(0, std::ios::end);  fsize = file.tellg();  file.seekg(0, std::ios::beg);  // Проверка типа файла архива  std::string type = "";  for (int i = 0; i < 4; i++) {  file.read((char\*)(&byte), sizeof(byte));  type += byte;  }  if (type != "GNEG") {  file.close();  return;  }  std::array<int, BYTE\_MAX> freq = {};  // Построение дерева  for (int i = 0; i < freq.size(); i++) {  file.read((char\*)(&freq[i]), sizeof(freq[i]));  hf\_tree.addByte(i, freq[i]); // i - байт, byte - частота  }  hf\_tree.makeTree();  // Создание изначального файла  std::ofstream out(outName, std::ios::binary);  HuffmanTree::Node\* node = hf\_tree.getRoot();  while (!file.eof() && file.read((char\*)(&byte), sizeof(byte))) {  for (int i = 0; i < 8; ++i) {  if (byte & (1 << (7 - i))) node = node->right;  else node = node->left;  if (!node->left && !node->right) {  out.write((char\*)(&node->c), sizeof(node->c));  node = hf\_tree.getRoot();  }  }  processed++;  updateProgress();  }  file.close();  }  // Сброс прогресса по завершению  processed = 0;  updateProgress();  } |
| huffman.cpp |
| #include <map>  #include <iostream>  #include "huffman.h"  /\*  \* Рекурсивная функция, строящая дерево Хаффмана  \*/  void HuffmanTree::makeTree(int& size) {  if (size > 1) {  int min1 = INT\_MAX, min2 = INT\_MAX;  int n1 = 0, n2 = 0;  for (int i = 0; i < BYTE\_MAX; i++)  if (trees[i] != nullptr && trees[i]->p < min1) {  n1 = i; min1 = trees[i]->p;  }  for (int i = 0; i < BYTE\_MAX; i++)  if (trees[i] != nullptr && trees[i]->p < min2 && i != n1) {  n2 = i; min2 = trees[i]->p;  }  Node\* tmp = new Node;  tmp->left = trees[n1]; tmp->right = trees[n2];  tmp->p = trees[n1]->p + trees[n2]->p;  tmp->parent = nullptr;  trees[n1]->parent = tmp; trees[n2]->parent = tmp;  trees[n1] = tmp; trees[n2] = nullptr;  makeTree(--size);  }  }  /\*  \* Рекурсивная функция, строящая коды для символов на основе дерева Хаффмана  \*/  void HuffmanTree::buildCodes(Node\* node, const std::string& code) {  if (!node) return;  if (node->p == 0) return;  // Если потомков нет - код найден  if (!node->left && !node->right) {  codes[node->c] = code;  }  buildCodes(node->left, code + "0");  buildCodes(node->right, code + "1");  }  HuffmanTree::HuffmanTree() {  for (int i = 0; i < BYTE\_MAX; i++) {  trees[i] = new Node;  trees[i]->c = i;  freq[i] = 0;  }  }  HuffmanTree::~HuffmanTree() {  codes.clear();  freq.fill(0);  //delete root;  }  /\*  \* Метод для добавления байта в дерево некоторое кол-во раз  \*/  void HuffmanTree::addByte(unsigned char byte, int count) {  trees[byte]->p += count;  freq[byte] += count;  }  /\*  \* Метод построения дерева Хаффмана, вызывающий рекурсивные  \* функции по построению дерева и нахождения кодов для байтов  \*/  void HuffmanTree::makeTree() {  int size = BYTE\_MAX;  makeTree(size);  for (int i = 0; i < BYTE\_MAX; i++)  if (trees[i] != nullptr) { root = trees[i]; break; };  buildCodes(root, "");  }  /\*  \* Метод для получения кода байта  \*/  std::string HuffmanTree::getCode(unsigned char byte) {  return codes[byte];  }  /\*  \* Метод для получения частот  \*/  std::array<int, BYTE\_MAX> HuffmanTree::getFreq() const {  return freq;  }  /\*  \* Метод для получения корня дерева  \*/  HuffmanTree::Node\* HuffmanTree::getRoot() {  return root;  }; |

Алгоритм работы:

Программа ожидает от пользователя путь к файлу из графического интерфейса. В зависимости от расширения файла, программа подсвечивает кнопки «Сжать» / «Разжать».

Сжатие происходит следующим образом:

Создаётся объект дерева Хаффмана, открывается запрошенный файл для чтения в бинарном виде. При помощи метода в дерево Хаффмана читается из файла и добавляется каждый байт входного файла. Затем вызывается метод построения дерева. Входной файл закрывается. После для бинарной записи открывается выходной файл (архив), в который записывается заголовок, состоящий из указателя на тип файла GNEG (4 байта) и массива частот байт (1024 байт), индекс которого определяет значение байта, а элемент массива определяет частоту (сам массив частот возвращается соответствующим методом из объекта дерева). Затем входной файл открывается для чтения и для каждого его байта по дереву Хаффмана определяется код, который затем делится на части по 8 бит и записывается в байт выходного архива. Процесс повторяется пока во входном файле не останется данных.

Разжатие происходит следующим образом: Открывается для бинарного чтения файл-архив и создаётся новый объект дерева Хаффмана. Читаются первые 4 байта из архива и проверяется на соответствие тип файла, затем из заголовка извлекается массив частот. По массиву частот в дерево вносится необходимое количество байт и вызывается метод по построению дерева. Открывается выходной файл для бинарной записи. Для каждого байта из архива происходит побитовый проход по дереву (0 – влево, 1 - вправо), после чего если код найден, значение байта поэтому записывается в выходной файл. Процесс повторяется пока в архиве не останется данных.

Результат работы:

