**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:      /\*      \* 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() {          archiver->compress(to\_string(filenameBox->Text), to\_string(filenameBox->Text + ".gzp"));      }      /\*      \* Метод-обёртка для выполнения разжатия в отдельном потоке      \*/      System::Void DecompressTask() {          archiver->compress(to\_string(filenameBox->Text), to\_string(filenameBox->Text + ".gzp"));      }      /\*      \* Метод, обновляющий прогрессбар      \*/      System::Void UpdateProgress(System::Object^ sender, System::EventArgs^ e) {          progressBar1->Value = archiver->getProgress();          if (progressBar1->Value == 100) progressBar1->Visible = false;      }      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];          // Если .gzp - можно разжать, если нет - нельзя          if (ext == "gzp") {              buttonComp->Enabled = false;              buttonUnComp->Enabled = true;          }          else {              buttonComp->Enabled = true;              buttonUnComp->Enabled = false;          }      }        System::Void buttonComp\_Click(System::Object^ sender, System::EventArgs^ e) {          if (filenameBox->Text != "") {              Task^ task = Task::Run(gcnew Action(this, &GUI::CompressTask));          }          progressBar1->Visible = true;      }      System::Void buttonUnComp\_Click(System::Object^ sender, System::EventArgs^ e) {          if (filenameBox->Text != "") {              Task^ task = Task::Run(gcnew Action(this, &GUI::DecompressTask));          }          progressBar1->Visible = true;      }  };  } |
| archiver.cpp |
| 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);          processed = 0;          // Создание дерева Хаффмана из байт          unsigned char byte;          while (file.read(reinterpret\_cast<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(reinterpret\_cast<const char\*>(&freq[i]), sizeof(freq[i]));          // Запись сжатых данных          file.open(fileName, std::ios::binary);          std::string buffer = "";          while (file.read(reinterpret\_cast<char\*>(&byte), sizeof(byte))) {              buffer += hf\_tree.getCode(byte);              while (buffer.size() >= 8) {                  unsigned char compByte = 0;                  for (int i = 0; i < 8; ++i)                      if (buffer[i] == '1')                          compByte += static\_cast<unsigned char>(pow(2, 7 - i));                  out.write(reinterpret\_cast<char\*>(&compByte), sizeof(compByte));                  buffer = buffer.substr(8);              }              processed++;              updateProgress();          }          file.close();          out.close();      }  }  /\*  \* 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);          processed = 0;          // Проверка типа файла архива          std::string type = "";          for (int i = 0; i < 4; i++) {              file.read(reinterpret\_cast<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(reinterpret\_cast<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(reinterpret\_cast<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(reinterpret\_cast<char\*>(&node->c), sizeof(node->c));                      node = hf\_tree.getRoot();                  }              }              processed++;              updateProgress();          }          file.close();      }  } |
| 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»/»Расжать».

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).

Результат работы: