|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 2.6** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Кодирование и сжатие данных методами без потерь»** | |
|  | |
| Выполнил студент группы ИКБО-30-22 | Сенькевич Г. Д. |
| Принял преподаватель | Красников С. А. |

Москва 2023

1. **Цель.** Получить знания и навыки работы с методами кодирования.
2. **Постановка задачи.**

Задание 1: Исследование алгоритмов сжатия на примерах

1) Выполнить каждую задачу варианта, представив алгоритм решения в виде таблицы и указав результат сжатия. Примеры оформления решения представлены в Приложении1 этого документа.

2) Описать процесс восстановления сжатого текста.

3) Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта.

Задание 2: разработать программы сжатия и восстановления текста методами Хаффмана и Шеннона – Фано.

1) Реализовать и отладить программы.

2) Сформировать отчет по разработке каждой программы в соответствии с требованиями. - По методу Шеннона-Фано привести: постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования, код и результаты тестирования. Рассчитать коэффициент сжатия. Сравнить с результат сжатия вашим алгоритмом с результатом людого архиватора. - по методу Хаффмана выполнить и отобразить результаты выполнения всех требований, предъявленных в задании и оформить разработку программы: постановка, подход к решению, код, результаты тестирования.

Вариант №20. Условие задания:

|  |  |  |
| --- | --- | --- |
| Закодировать фразу методами Шеннона– Фано | Сжатие данных по методу Лемпеля– Зива LZ77 Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу: | Закодировать следующую фразу, используя код LZ78 |
| Эни бэни рики паки Турбаурбасентибряки  .  Может – выйдет, может – нет, В общем – полный Интернет | 0100101010010000101 | mantopmentopomantomen |

# **Решение**

Задание 1:

1. Для метода Шеннон – Фано

Закодировать фразу

«Эни бэни рики паки Турбаурбасентибряки . Может – выйдет, может – нет, В общем – полный Интернет»,

используя метод Шеннона–Фено.

Этапы построения кодов Шеннона-Фано:

1. Расположить все символы алфавита в порядке невозрастания вероятностей.

2. Разделить весь алфавит на две группы так, чтобы суммарные вероятности в каждой из групп были примерно равны. Первой или верхней группе в качестве первого символа кодовой комбинации присвоить 1, второй или нижней – 0.

3. Продолжаем процесс разделения: каждую из групп вновь делим на две подгруппы с примерно одинаковыми вероятностями. В качестве вторых символов кодовых комбинаций для верхних подгрупп выбираем 0, а для нижних – 1.

4. Весь процесс повторяется до тех пор, пока в каждой из подгрупп не останется по одному символу (листы дерева).

Декодирование:

1. Закодированный текст считывается бит за битом.
2. Начиная с корня дерева, для каждого бита выбирается соответствующая ветвь, и процесс продолжается до тех пор, пока не достигнут лист дерева.
3. Когда достигнут лист, соответствующий символ из оригинального текста восстанавливается, и процесс продолжается для следующих битов.
4. Процесс декодирования продолжается для каждого бита закодированного текста, пока весь текст не будет восстановлен.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Символ | количество | 1я цифра | 2я цифра | 3я цифра | 4я цифра | 5я цифра | 6я цифра | 7я  цифра | код | количество бит |
| " " | 16 | 0 | 0 | 0 |  |  |  |  | 000 | 48 |
| и | 8 | 0 | 0 | 1 |  |  |  |  | 001 | 24 |
| т | 8 | 0 | 1 | 0 |  |  |  |  | 010 | 24 |
| е | 8 | 0 | 1 | 1 | 0 |  |  |  | 0110 | 32 |
| н | 7 | 0 | 1 | 1 | 1 |  |  |  | 0111 | 28 |
| Б | 5 | 1 | 0 | 0 | 0 |  |  |  | 1000 | 20 |
| Р | 5 | 1 | 0 | 0 | 1 |  |  |  | 1001 | 20 |
| о | 4 | 1 | 0 | 1 | 0 | 0 |  |  | 10100 | 20 |
| к | 3 | 1 | 0 | 1 | 0 | 1 |  |  | 10101 | 15 |
| а | 3 | 1 | 0 | 1 | 1 | 0 |  |  | 10110 | 15 |
| м | 3 | 1 | 0 | 1 | 1 | 1 |  |  | 10111 | 15 |
| - | 3 | 1 | 1 | 0 | 0 | 0 |  |  | 11000 | 15 |
| э | 2 | 1 | 1 | 0 | 0 | 1 | 0 |  | 110010 | 12 |
| п | 2 | 1 | 1 | 0 | 0 | 1 | 1 |  | 110011 | 12 |
| у | 2 | 1 | 1 | 0 | 1 | 0 |  |  | 11010 | 10 |
| ж | 2 | 1 | 1 | 0 | 1 | 1 | 0 |  | 110110 | 12 |
| в | 2 | 1 | 1 | 0 | 1 | 1 | 1 |  | 110111 | 12 |
| ы | 1 | 1 | 1 | 1 | 0 | 0 |  |  | 11100 | 5 |
| й | 1 | 1 | 1 | 1 | 0 | 1 | 0 |  | 111010 | 6 |
| , | 1 | 1 | 1 | 1 | 0 | 1 | 1 |  | 111011 | 6 |
| с | 1 | 1 | 1 | 1 | 1 | 0 | 0 |  | 111100 | 6 |
| я | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1111010 | 7 |
| . | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1111011 | 7 |
| д | 1 | 1 | 1 | 1 | 1 | 1 | 0 |  | 111110 | 6 |
| щ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1111110 | 7 |
| л | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1111111 | 7 |

Таблица 1. Метод Шеннона-Фано

Незакодированная фраза – 97\*8 = 776 бит. Закодированная – 391 бит.

1. Для метода LZ77 - Закодировать фразу «0100101010010000101»

|  |  |
| --- | --- |
| Исходный текст | 0100101010010000101 |
| 0. 1. 00. 10. 101. 001. 000. 0101. |
| LZ-КОД | 0 1 001 100 011 1010 1101 1001 1001 001 |
| R | 2 3 4 |
| Вводимые коды | - 10 11 100 101 110 111 1000 1001 1010 |

Таблица 2. Метод LZ77

Где LZ – сжатый текст (в данном примере в связи с небольшим размером исходного текста размер текста не уменьшился) R отмечает шаги кодирования, после которых происходит переход на представление кодов А увеличенным числом разрядов R. Так, на первом шаге вводится код 10 для комбинации 00, и поэтому на следующих двух шагах R = 2, после третьего шага R = 3, после седьмого шага R = 4, т.е. в общем случае R = K после шага 2K-1 –1.

1. Для метода LZ78 - Закодировать фразу «долделдолдилделдил»

В отличие от LZ77, работающего с уже полученными данными, LZ78 ориентируется на данные, которые только будут получены (LZ78 не использует скользящее окно, он хранит словарь из уже просмотренных фраз). Алгоритм считывает символы сообщения до тех пор, пока накапливаемая подстрока входит целиком в одну из фраз словаря. Как только эта строка перестанет соответствовать хотя бы одной фразе словаря, алгоритм генерирует код, состоящий из индекса строки в словаре, которая до последнего введенного символа содержала входную строку, и символа, нарушившего совпадение. Затем в словарь добавляется введенная подстрока. Если словарь уже заполнен, то из него предварительно удаляют менее всех используемую в сравнениях фразу. Если в конце алгоритма мы не находим символ, нарушивший совпадения, то тогда мы выдаем код в виде (индекс строки в словаре без последнего символа, последний символ). Словарь хранится в префиксном дереве, что позволяет легко находить самое длинное продолжение входной строки, уже присутствующее в словаре. При декодировании строится в точности этот же словарь. В сжатом представлении строки словарь не хранится.

Декодирование:

1. Имеем закодированную последовательность пар (индекс, символ).
2. Начинаем с первой пары и поочередно обрабатываем каждую.
   * Для каждой пары (индекс, символ):
     + Если индекс равен 0, это означает, что символ является новым, и добавляем его к результату.
     + Если индекс не равен 0, это означает, что символ формируется путем объединения символов, предшествующих данной паре. По индексу находим соответствующую подстроку в результате и добавляем к ней текущий символ.
3. Процесс продолжается до обработки всей закодированной последовательности.
4. Результатом декодирования является оригинальный текст, который был сжат методом LZ78.

|  |  |  |
| --- | --- | --- |
| **Словарь** | **Содержимое считываемой строчки** | **Код** |
|  | Д | <0, Д> |
| Д = 1 | О | <0, О> |
| Д = 1, O = 2 | Л | <0, Л> |
| Д = 1, O = 2, Л = 3 | ДЕ | <1, Е> |
| Д = 1, O = 2, Л = 3, ДЕ = 4 | ЛД | <3, Д> |
| Д = 1, O = 2, Л = 3, ДЕ = 4, ЛД = 5 | ОЛ | <2, Л> |
| Д = 1, O = 2, Л = 3, ДЕ = 4, ЛД = 5, ОЛ = 6 | ДИ | <1, И> |
| Д = 1, O = 2, Л = 3, ДЕ = 4, ЛД = 5, ОЛ = 6, ДИ = 7 | ЛДЕ | <5, Е> |
| Д = 1, O = 2, Л = 3, ДЕ = 4, ЛД = 5, ОЛ = 6, ДИ = 7, ЛДЕ = 8 | ЛДИ | <5, И> |
| Д = 1, O = 2, Л = 3, ДЕ = 4, ЛД = 5, ОЛ = 6, ДИ = 7, ЛДЕ = 8, ЛДИ = 9 |  | <3, EOF> |

Таблица 3. Метод LZ78

Задание 2:

* + 1. Алгоритм Шеннона – Фано.

Алгоритм Шеннона-Фано представляет собой один из методов сжатия данных, разработанный Клодом Шенноном и Робертом Фано в 1949 году. Этот алгоритм основывается на идее разделяющего кодирования, где символы входной последовательности разделяются на две группы с приблизительно равными вероятностями. Процесс работы алгоритма включает в себя рекурсивное разбиение символов на подгруппы до достижения определенного критерия разделения. Каждой подгруппе присваивается уникальный битовый код, который строится в процессе разделения. В результате получается переменная длина кодов для каждого символа, что позволяет сжимать данные в зависимости от их статистических характеристик. Алгоритм Шеннона-Фано широко применяется для сжатия данных, особенно в тех случаях, когда вероятности символов значительно различаются. Его эффективность основана на том, что часто встречающиеся символы получают более короткие коды, что уменьшает общую длину закодированной последовательности.

Для реализации алгоритма Шеннона – Фано были созданы следующие функции и структуры:

* Структура shanonFanoNode определяет узел для хранения информации о символе, его вероятности и коде.

|  |
| --- |
| // Структура для представления символа, его вероятности и кода  struct shanonFanoNode {  char symbol; // Символ  double probability; // Вероятность символа  string code; // Код символа  }; |

* compareSymbols — функция сравнения для упорядочивания символов в векторе по убыванию вероятности.

|  |
| --- |
| // Функция для сравнения символов по вероятности в убывающем порядке  bool compareSymbols(SymbolNode a, SymbolNode b) {  return a.probability > b.probability;  } |

* encodeText — рекурсивная функция для генерации кодов Шеннона-Фано. Делит вектор символов на две части, присваивая им бинарные коды, и затем рекурсивно вызывает сама себя для каждой из частей.

|  |
| --- |
| void encodeText(int begin, int end, vector<SymbolNode>& frequencyTable) {  // Если разность индексов равна 1, завершаем выполнение функции  if (end - begin == 1) {  return;  }  int separator = begin + 1; // Устанавливаем разделитель  double sumLeft = frequencyTable[begin].probability; // Сумма вероятностей слева  double sumRight = 0; // Сумма вероятностей справа  // Считаем сумму вероятностей справа от разделителя  for (int i = separator; i < end; i++) {  sumRight += frequencyTable[i].probability;  }  // Пока сумма вероятностей слева меньше суммы вероятностей справа  while (separator < end - 1 && sumLeft < sumRight) {  sumLeft += frequencyTable[separator].probability; // Добавляем вероятность элемента к сумме слева  separator++; // Сдвигаем разделитель  sumRight -= frequencyTable[separator].probability; // Убираем вероятность элемента из суммы справа  }  // Добавляем к коду символов слева '0', справа '1'  for (int i = begin; i < separator; i++) {  frequencyTable[i].code += "0";  }  for (int i = separator; i < end; i++) {  frequencyTable[i].code += "1";  }  // Рекурсивно вызываем функцию для левой и правой части таблицы  ShannonFanoGenerate(begin, separator, frequencyTable); // Левая часть  ShannonFanoGenerate(separator, end, frequencyTable); // Правая часть  } |

* decodeText — функция декодирования текста. Использует таблицу символов и закодированный текст для восстановления исходного текста.

|  |
| --- |
| // Функция для декодирования текста  string decodeText(vector<SymbolNode>& frequencyTable, string encodedText) {  string text = "";  while (!encodedText.empty()) {  for (auto& symbol : frequencyTable) {  // Проверяем, начинается ли закодированный текст с кода символа  if (encodedText.rfind(symbol.code, 0) == 0) {  text += symbol.symbol;  encodedText.erase(0, symbol.code.size());  break;  }  }  }  return text;  } |

* shannonFanoMain — основная функция для работы с кодами Шеннона-Фано. Сначала собирает частоты символов, создает таблицу символов, сортирует их по вероятности, генерирует коды Шеннона-Фано и выводит закодированный текст, декодированный текст, количество символов в исходном тексте, вес закодированного и изначального текста в битах, а также процент сжатия.

|  |
| --- |
| // Основная функция для работы с кодами Шеннона-Фано  void shannonFanoMain(string text) {  map<char, int> symbolFrequencies; // Карта для хранения частот каждого символа  for (char ch : text) {  symbolFrequencies[ch]++;  }  // создание таблицы символов  vector<SymbolNode> frequencyTable;  for (auto it : symbolFrequencies) {  SymbolNode node;  node.symbol = it.first;  node.probability = (double)it.second / text.length();  frequencyTable.push\_back(node);  }  sort(frequencyTable.begin(), frequencyTable.end(), CompareSymbols);  ShannonFanoGenerate(0, frequencyTable.size(), frequencyTable);  float totalBitCount = 0;  for (SymbolNode symbol : frequencyTable) {  int totalCount = symbol.probability \* text.length();  totalBitCount += totalCount \* symbol.code.length();  }    string encodedText = "";  for (char ch : text) {  for (SymbolNode symbol : frequencyTable) {  if (ch == symbol.symbol) {  encodedText += symbol.code;  break;  }  }  }  cout << "Закодированный текст: " << encodedText << endl;  string decodedText = DecodeShannonFano(frequencyTable, encodedText);  cout << "Декодированный текст: " << decodedText << endl;  cout << "Количество символов в тексте: " << text.length() << endl;  cout << "Итоговый вес закодированного текста в битах: " << int(totalBitCount) << " бит" << endl;  float totalBitText = text.length() \* 8;  cout << "Вес изначального текста в битах: " << int(totalBitText) << " бит" << endl;  float compressionPercentage = 100 \* totalBitCount / totalBitText;  cout << "Процент сжатия: " << 100 - compressionPercentage << "%" << endl;  } |

* + 1. Провести кодирование(сжатие) исходной строки символов «Черных Тимофей Юрьевич» с использованием алгоритма Хаффмана
* Была построена таблица частот встречаемости символов в исходной строке символов для чего был сформирован алфавит исходной строки и посчитано количество вхождений (частот) символов и их вероятности появления, например, для строки Черных Тимофей Юрьевич такая отсортированная таблица будет иметь вид:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| алфавит | Е | Р | И | " " | Ч | Ы | Х | Т |
| количество вх | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
| вероятность | 0,14 | 0,09 | 0,09 | 0,09 | 0,09 | 0,05 | 0,05 | 0,05 |
| алфавит | Н | М | О | Ф | Й | Ю | Ь | В |
| количество вх | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| вероятность | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 |

Таблица 4. Таблица частот

* Было построено дерево кодирования Хаффмана, в данном примере оно имеет вид:

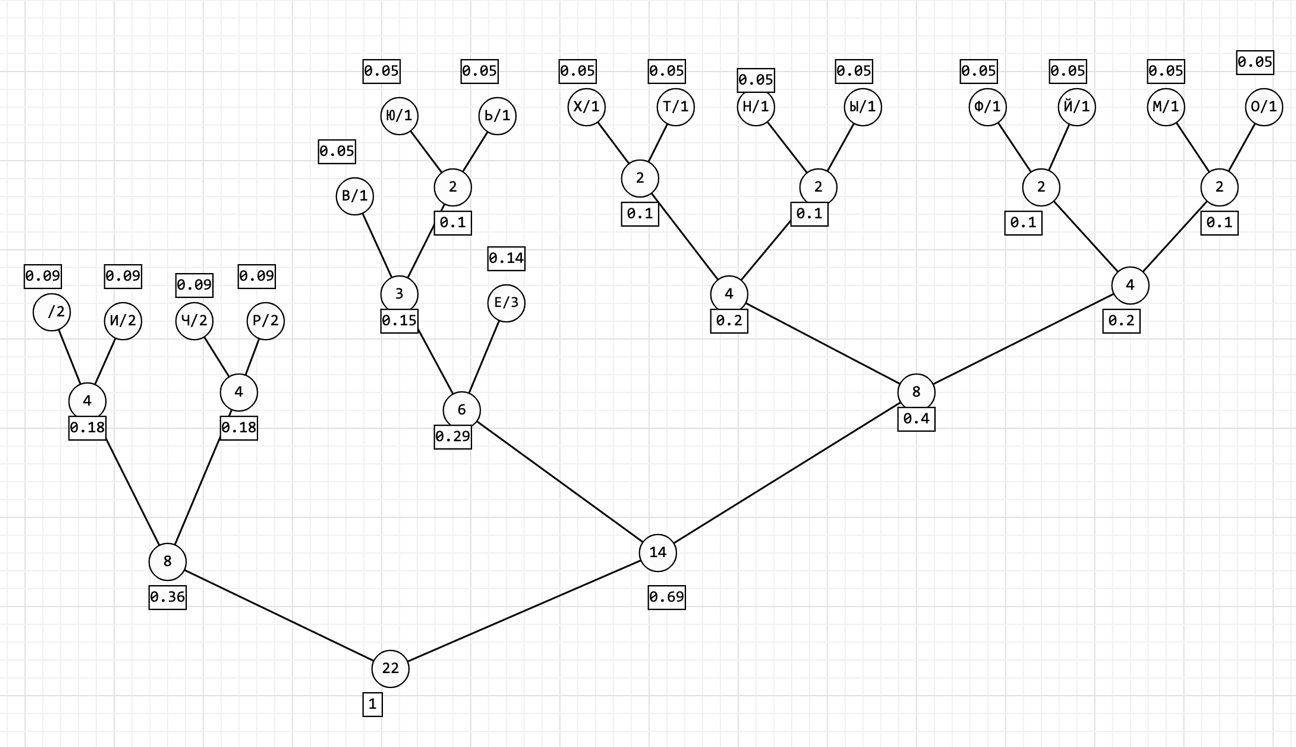


Рисунок 1. Древо кодирования Хафмана

* Каждой ветке был присвоен свой код, после упорядочивания дерева:

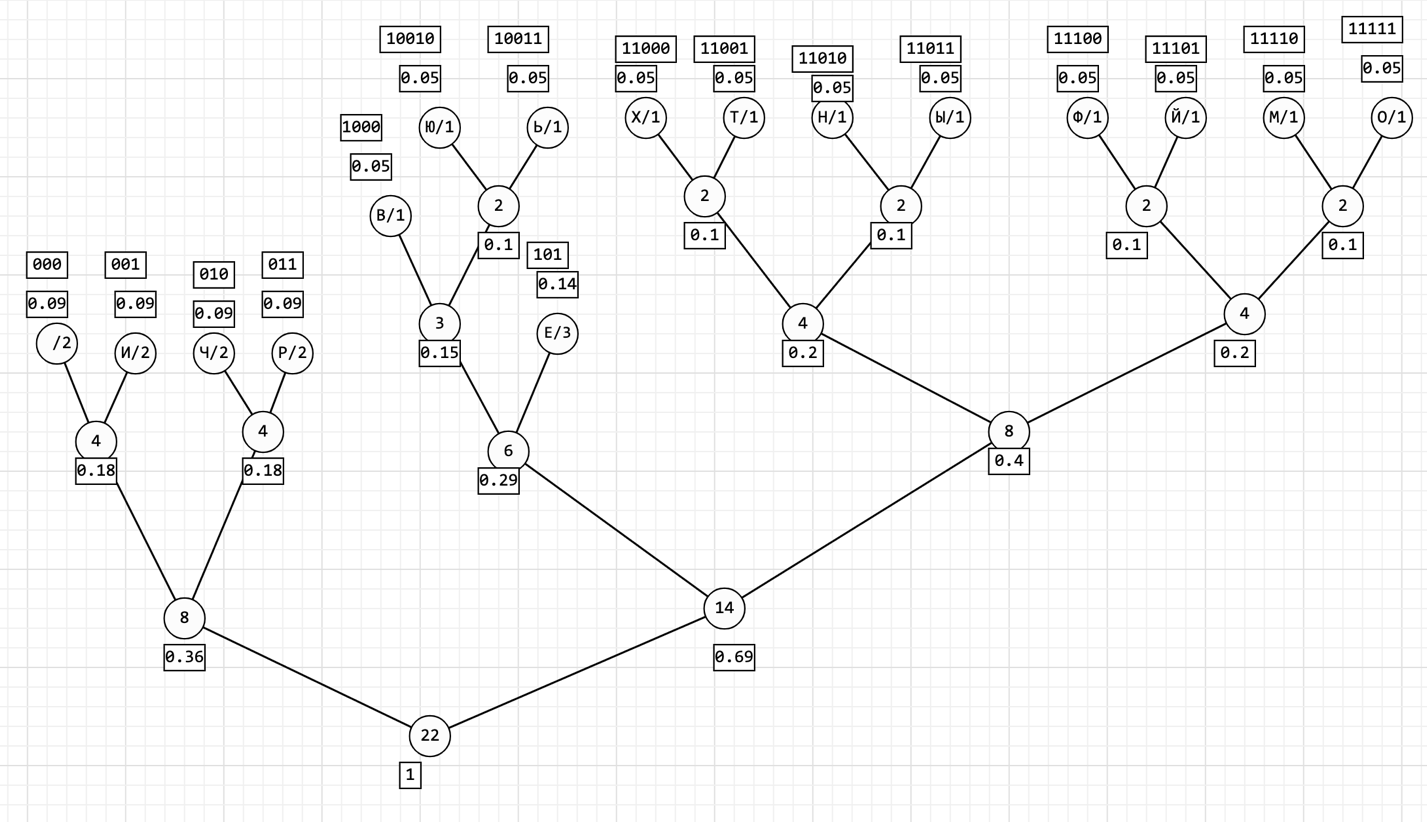


Рисунок 2. Древо кодирования Хафмана с кодами

* После было проведено кодирование исходной строки:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ч | Е | Р | | Н | | Ы | | Х | | | “ “ | | Т | | И | | | М | |
| 010 | 101 | 011 | | 11010 | | 11011 | | 11000 | | | 000 | | 11001 | | 001 | | | 11110 | |
|  |  |  | |  | |  | |  | | |  | |  | |  | | |  | |
| О | Ф | | Е | | Й | | “ “ | | Ю | Р | | Ь | | Е | | В | И | | Ч | |
| 11111 | 11100 | | 101 | | 11101 | | 000 | | 10010 | 011 | | 10011 | | 101 | | 1000 | 001 | | 010 | |

Таблица 5. Итоговое кодирование

* Незакодированная фраза – 22\*8 = 146 бит, закодированная –3\*11+5\*10+4=87 бит
* Коэффициент сжатия – 2.023
* Коэффициент равномерного сжатия –1.264
* Средняя длина полученного кода – 4.27
* Дисперсия – 1.08

Решение данной задачи кодом представлено в тестировании

3. Применить алгоритм Хаффмана для архивации данных текстового файла.

Алгоритм Хаффмана, предложенный Дэвидом Хаффманом в 1952 году, представляет собой эффективный метод сжатия данных. Этот алгоритм основан на принципе переменной длины кодирования, где символы входной последовательности кодируются битовыми строками различной длины в зависимости от их частоты встречаемости. Основная идея заключается в том, чтобы присвоить более короткие битовые коды часто встречающимся символам, а более длинные коды редким символам. Это позволяет уменьшить общую длину закодированной последовательности и, следовательно, выполнить сжатие данных. Процесс построения кодов Хаффмана включает в себя построение двоичного дерева, в листьях которого находятся символы с их частотами. Затем коды символов строятся на основе пути от корня дерева к каждому листу. Символы, встречающиеся чаще, получают более короткие коды, что делает алгоритм эффективным для сжатия данных с учетом их статистических характеристик. Алгоритм Хаффмана широко используется в области сжатия данных, включая архивацию файлов, передачу данных по сети и другие приложения, где эффективное уменьшение объема информации играет важную роль.

Для реализации алгоритма Хаффмана были созданы следующие функции и структуры:

* HuffmanNode — структура, используемая для представления узла в дереве Хаффмана. Каждый узел содержит символ (data), его частоту в тексте (frequency), и указатели на левого и правого потомков (left и right). Конструктор узла принимает символ и его частоту при создании узла.

|  |
| --- |
| // Структура для представления узла в дереве Хаффмана  struct HuffmanNode {  char data; // Символ узла  int frequency; // Частота символа в тексте  HuffmanNode\* left; // Указатель на левого потомка  HuffmanNode\* right; // Указатель на правого потомка  // Конструктор для удобного создания узла с данными и частотой  HuffmanNode(char data, int frequency) : data(data), frequency(frequency), left(nullptr), right(nullptr) {}  }; |

* CompareNodes — функтор, определенный для сравнения двух узлов дерева Хаффмана. Он используется для сортировки узлов в приоритетной очереди, где узлы с более низкой частотой имеют более высокий приоритет.

|  |
| --- |
| struct CompareNodes {  bool operator()(HuffmanNode\* left, HuffmanNode\* right) {  return left->frequency > right->frequency;  }  }; |

* generateCodes — рекурсивная функция, которая проходит по дереву Хаффмана, начиная с корня, и присваивает коды листьям. Код формируется добавлением "0" для левого поддерева и "1" для правого поддерева. Полученные коды сохраняются в словаре codes, используемом для последующего кодирования текста. Общая сложность генерации кодов - O(n).

|  |
| --- |
| // Рекурсивная функция для генерации кодов Хаффмана  void generateCodes(HuffmanNode\* root, string code, unordered\_map<char, string>& codes) {  if (root) {  if (!root->left && !root->right) {  codes[root->data] = code; // Если узел является листом, добавляем код в словарь  }  generateCodes(root->left, code + "0", codes); // Рекурсивно обрабатываем левое поддерево с добавлением "0" к текущему коду  generateCodes(root->right, code + "1", codes); // Рекурсивно обрабатываем правое поддерево с добавлением "1" к текущему коду  }  } |

* buildHuffmanTree — функция, строящая дерево Хаффмана и создающая коды символов. Сначала создается частотная карты для каждого символа в тексте. Затем строится приоритетная очередь из узлов, представляющих символы и их частоты. Узлы объединяются до тех пор, пока не останется единственный корневой узел. Наконец, вызывается generateCodes для получения кодов символов. Сложность построения дерева - O(n log n).

|  |
| --- |
| HuffmanNode\* buildHuffmanTree(string text, unordered\_map<char, string>& codes) {  unordered\_map<char, int> frequencyMap;  for (char ch : text) {  frequencyMap[ch]++;  }  // Приоритетная очередь для узлов с приоритетом наименьшей частоты  priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, CompareNodes> minHeap;  for (auto entry : frequencyMap) {  HuffmanNode\* node = new HuffmanNode(entry.first, entry.second);  minHeap.push(node);  }  // Строим дерево Хаффмана  while (minHeap.size() > 1) {  HuffmanNode\* left = minHeap.top();  minHeap.pop();  HuffmanNode\* right = minHeap.top();  minHeap.pop();  HuffmanNode\* mergedNode = new HuffmanNode('$', left->frequency + right->frequency);  mergedNode->left = left;  mergedNode->right = right;  minHeap.push(mergedNode);  }  // Получаем коды Хаффмана  HuffmanNode\* root = minHeap.top();  generateCodes(root, "", codes);  return root;  } |

* encodeText — функция, кодирующая текст с использованием предварительно созданных кодов Хаффмана. Проходит по каждому символу в тексте и конкатенирует его соответствующий код. Сложность кодирования - O(m), где m - длина закодированного текста.

|  |
| --- |
| string encodeText(string text, unordered\_map<char, string>& codes) {  string encodedText = "";  for (char ch : text) {  encodedText += codes[ch];  }  return encodedText;  } |

* decodeText — функция, декодирующая текст с использованием дерева Хаффмана. Проходит по каждому биту в закодированном тексте и перемещается в соответствующие узлы дерева. Когда достигается лист дерева, добавляет символ к декодированному тексту и возвращается в корень дерева. Сложность кодирования - O(m), где m - длина закодированного текста.

|  |
| --- |
| // Функция для декодирования текста с использованием дерева Хаффмана  string decodeText(HuffmanNode\* root, string encodedText) {  string decodedText = "";  HuffmanNode\* current = root;  for (char bit : encodedText) {  if (bit == '0') {  current = current->left;  } else {  current = current->right;  }  if (!current->left && !current->right) {  decodedText += current->data;  current = root;  }  }  return decodedText;  } |

* huffmanMain — главная функция, объединяющая все этапы алгоритма Хаффмана. Строит дерево, кодирует и декодирует текст, затем выводит результаты, такие как декодированный текст, закодированный текст, вес изначального и закодированного текстов в битах, процент сжатия и коэффициент сжатия.

|  |
| --- |
| // Основная функция для работы с алгоритмом Хаффмана  void huffmanMain(string text) {  unordered\_map<char, string> codes;  HuffmanNode\* root = buildHuffmanTree(text, codes);  // Закодировать текст  string encodedText = encodeText(text, codes);  // Расчет процента и коэффициента сжатия  double originalSize = text.size() \* 8.0; // каждый символ представлен 8 битами  double compressedSize = encodedText.size();  double compressionRatio = originalSize / compressedSize;  double compressionPercentage = (1 - (compressedSize / originalSize)) \* 100;  // Декодировать текст  string decodedText = decodeText(root, encodedText);  // Вывод результатов  cout << "Декодированный текст: " << decodedText << endl;  cout << "Закодированный текст: " << encodedText << endl;  cout << "Итоговый вес закодированного текста в битах: " << compressedSize << " бит" << endl;  cout << "Вес изначального текста в битах: " << originalSize << " бит" << endl;  cout << "Процент сжатия: " << compressionPercentage << "%" << endl;  cout << "Коэффициент сжатия: " << compressionRatio << endl;  } |

Общая сложность алгоритма Хаффмана в данной реализации - O(n log n), где n - количество уникальных символов в тексте. Это эффективный алгоритм для сжатия текстовых данных и обладает линейной сложностью относительно размера входных данных.

# **Тестирование**

При запуске программы открывается пользовательский интерфейс, который позволяет автоматически заполнить таблицу, выполнить все возможные методы или запустить функцию тестирования программы

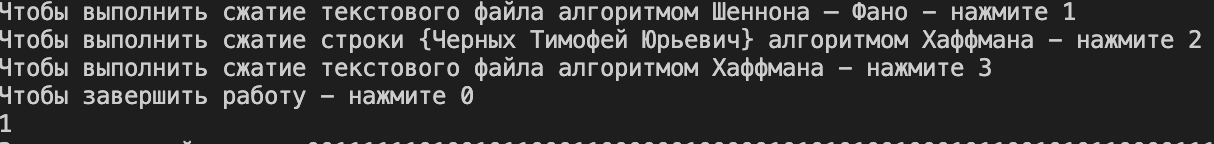


Рисунок 3. Пользовательский интерфейс программы.

Выполним тестирование выполнения первого задания. Как видно, программа кодирует и декодирует текст алгоритмом Шенона-Фано, выводит процент сжатия.

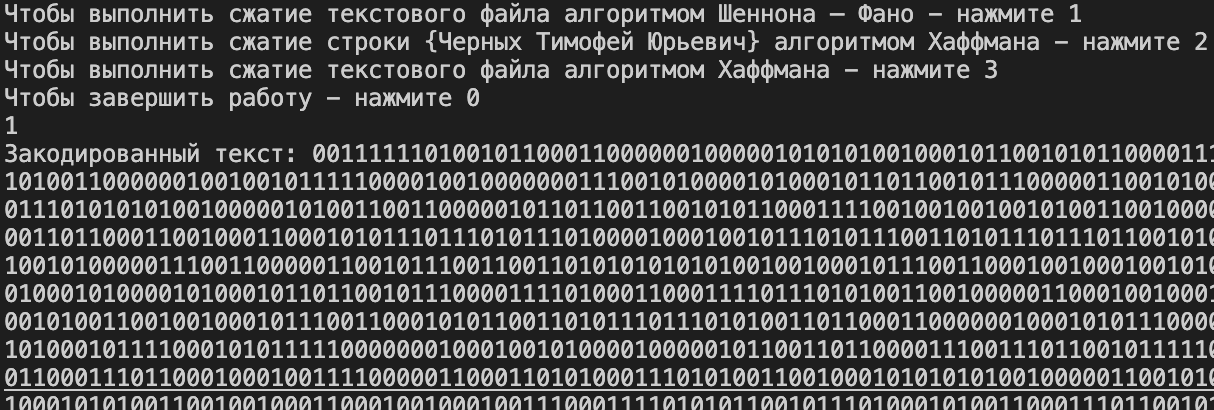


Рисунок 4. Выполнение первого задания

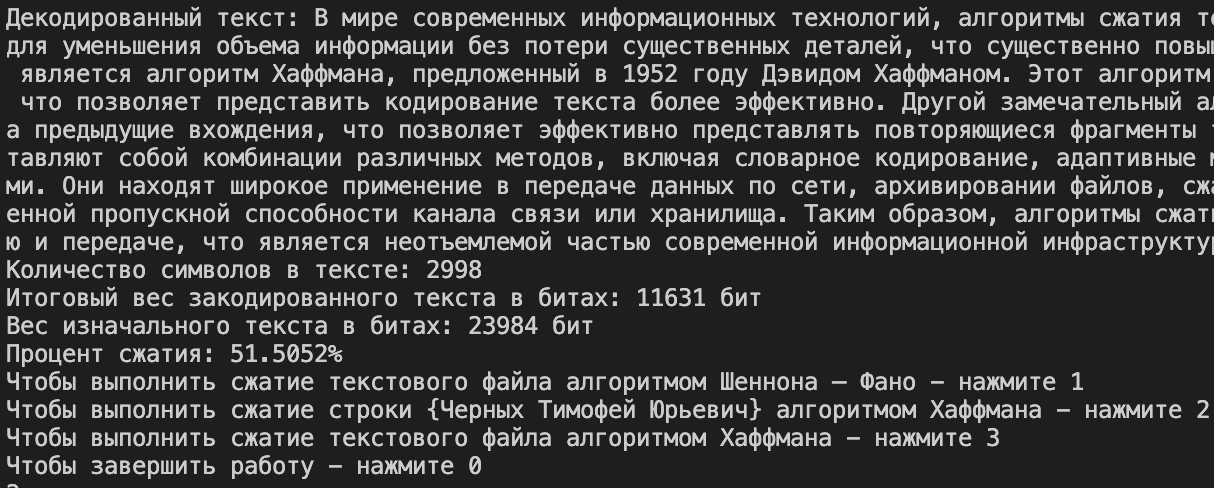


Рисунок 5. Выполнение первого задания

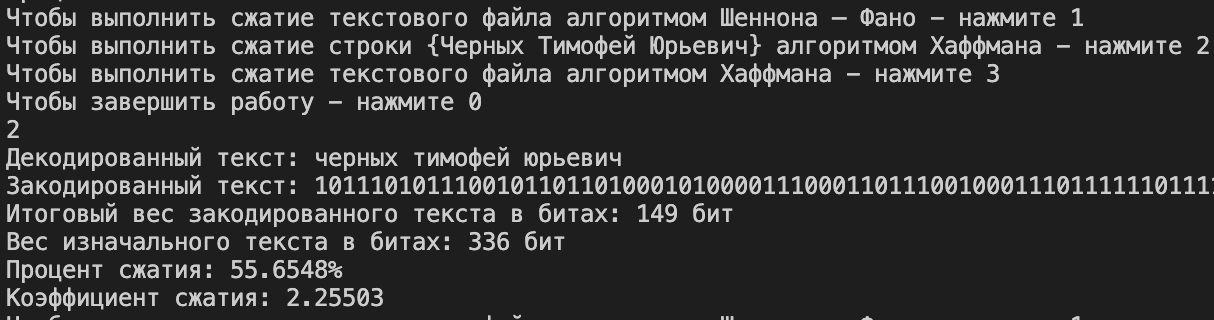
Выполним тестирование выполнения второго задания. Как видно, программа кодирует и декодирует текст алгоритмом Хаффмана, выводит процент и коэффициентсжатия.

Рисунок 6. Выполнение второго задания

Выполним тестирование выполнения третьего задания. Как видно, программа кодирует и декодирует текст алгоритмом Хаффмана, выводит процент и коэффициент сжатия.

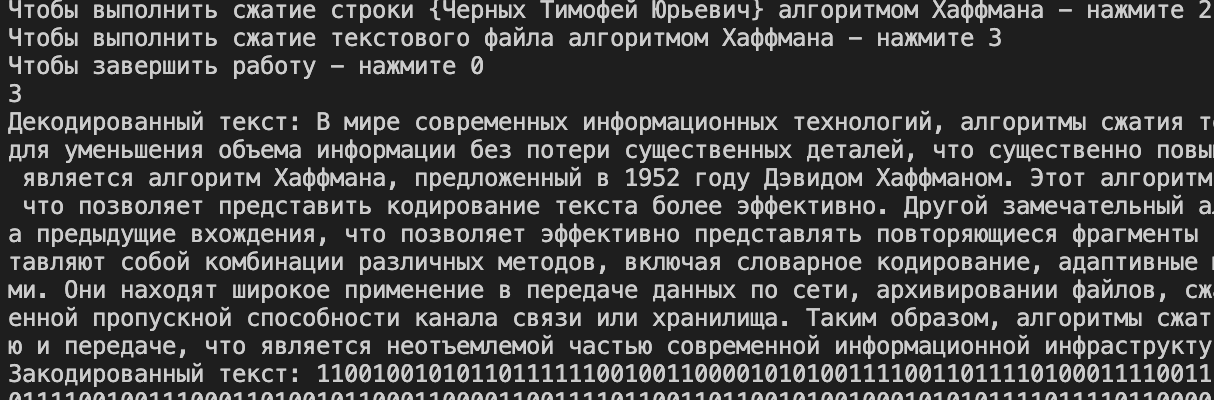


Рисунок 7. Выполнение третьего задания

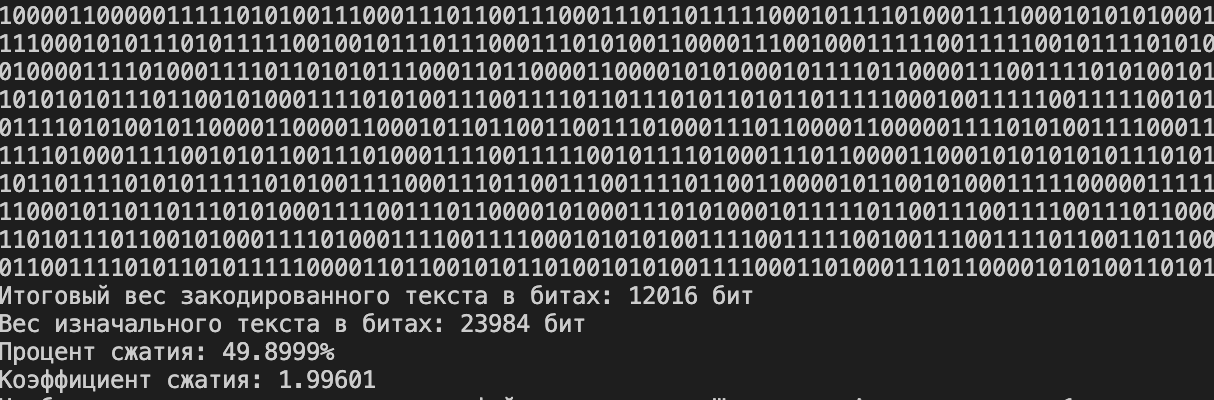


Рисунок 8. Выполнение третьего задания

Так же я выполнил сжатие текстового файла встроенным архиватором. Его размер оказался 12080 бит, что сопоставимо с сжатием алгоритмами Хаффмана и Шенона-Фано. Можно сделать вывод, что самым эффективным алгоритмом является алгоритм Шенона-Фано.

Из результатов выполнения программы видно:

1. Программа работает корректно, выполняя все поставленные задачи.

# **Вывод**

В результате выполнения работы я:

* Получил знания и навыки реализации алгоритмов сжатия текста, таких как: алгоритм Шенона-Фано, алгоритм Хаффмана, LZ77 и LZ78.

# **Исходный код программы**

Main.cpp

|  |
| --- |
| #include <iostream>  #include <stdlib.h>  #include <stdio.h>  #include <iostream>  #include <fstream>  #include <algorithm>  #include <regex>  #include <chrono>  #include <cstdlib>  #include <bitset>  #include <queue>  #include <map>  #include <unordered\_map>  #include <locale>  using namespace std::chrono;  using namespace std;  // Структура для представления символа, его вероятности и кода  struct SymbolNode {  char symbol; // Символ  double probability; // Вероятность символа  string code; // Код символа  };  // Функция для сравнения символов по вероятности в убывающем порядке  bool CompareSymbols(SymbolNode a, SymbolNode b) {  return a.probability > b.probability;  }  // Рекурсивная функция для генерации кодов Шеннона-Фано  void ShannonFanoGenerate(int begin, int end, vector<SymbolNode>& frequencyTable) {  // Если разность индексов равна 1, завершаем выполнение функции  if (end - begin == 1) {  return;  }  int separator = begin + 1; // Устанавливаем разделитель  double sumLeft = frequencyTable[begin].probability; // Сумма вероятностей слева  double sumRight = 0; // Сумма вероятностей справа  // Считаем сумму вероятностей справа от разделителя  for (int i = separator; i < end; i++) {  sumRight += frequencyTable[i].probability;  }  // Пока сумма вероятностей слева меньше суммы вероятностей справа  while (separator < end - 1 && sumLeft < sumRight) {  sumLeft += frequencyTable[separator].probability; // Добавляем вероятность элемента к сумме слева  separator++; // Сдвигаем разделитель  sumRight -= frequencyTable[separator].probability; // Убираем вероятность элемента из суммы справа  }  // Добавляем к коду символов слева '0', справа '1'  for (int i = begin; i < separator; i++) {  frequencyTable[i].code += "0";  }  for (int i = separator; i < end; i++) {  frequencyTable[i].code += "1";  }  // Рекурсивно вызываем функцию для левой и правой части таблицы  ShannonFanoGenerate(begin, separator, frequencyTable); // Левая часть  ShannonFanoGenerate(separator, end, frequencyTable); // Правая часть  }  // Функция для декодирования текста  string DecodeShannonFano(vector<SymbolNode>& frequencyTable, string encodedText) {  string text = "";  while (!encodedText.empty()) {  for (auto& symbol : frequencyTable) {  // Проверяем, начинается ли закодированный текст с кода символа  if (encodedText.rfind(symbol.code, 0) == 0) {  text += symbol.symbol;  encodedText.erase(0, symbol.code.size());  break;  }  }  }  return text;  }  // Основная функция для работы с кодами Шеннона-Фано  void ShannonFanoMain(string text) {  map<char, int> symbolFrequencies; // Карта для хранения частот каждого символа  for (char ch : text) {  symbolFrequencies[ch]++;  }  // создание таблицы символов  vector<SymbolNode> frequencyTable;  for (auto it : symbolFrequencies) {  SymbolNode node;  node.symbol = it.first;  node.probability = (double)it.second / text.length();  frequencyTable.push\_back(node);  }  sort(frequencyTable.begin(), frequencyTable.end(), CompareSymbols);  ShannonFanoGenerate(0, frequencyTable.size(), frequencyTable);  float totalBitCount = 0;  for (SymbolNode symbol : frequencyTable) {  int totalCount = symbol.probability \* text.length();  totalBitCount += totalCount \* symbol.code.length();  }    string encodedText = "";  for (char ch : text) {  for (SymbolNode symbol : frequencyTable) {  if (ch == symbol.symbol) {  encodedText += symbol.code;  break;  }  }  }  cout << "Закодированный текст: " << encodedText << endl;  string decodedText = DecodeShannonFano(frequencyTable, encodedText);  cout << "Декодированный текст: " << decodedText << endl;  cout << "Количество символов в тексте: " << text.length() << endl;  cout << "Итоговый вес закодированного текста в битах: " << int(totalBitCount) << " бит" << endl;  float totalBitText = text.length() \* 8;  cout << "Вес изначального текста в битах: " << int(totalBitText) << " бит" << endl;  float compressionPercentage = 100 \* totalBitCount / totalBitText;  cout << "Процент сжатия: " << 100 - compressionPercentage << "%" << endl;  }  //  // Структура для представления узла в дереве Хаффмана  struct HuffmanNode {  char data; // Символ узла  int frequency; // Частота символа в тексте  HuffmanNode\* left; // Указатель на левого потомка  HuffmanNode\* right; // Указатель на правого потомка  // Конструктор для удобного создания узла с данными и частотой  HuffmanNode(char data, int frequency) : data(data), frequency(frequency), left(nullptr), right(nullptr) {}  };  // Функтор для сравнения узлов в приоритетной очереди по частоте  struct CompareNodes {  bool operator()(HuffmanNode\* left, HuffmanNode\* right) {  return left->frequency > right->frequency;  }  };  // Рекурсивная функция для генерации кодов Хаффмана  void generateCodes(HuffmanNode\* root, string code, unordered\_map<char, string>& codes) {  if (root) {  if (!root->left && !root->right) {  codes[root->data] = code; // Если узел является листом, добавляем код в словарь  }  generateCodes(root->left, code + "0", codes); // Рекурсивно обрабатываем левое поддерево с добавлением "0" к текущему коду  generateCodes(root->right, code + "1", codes); // Рекурсивно обрабатываем правое поддерево с добавлением "1" к текущему коду  }  }  // Функция для построения дерева Хаффмана и получения кодов  HuffmanNode\* buildHuffmanTree(string text, unordered\_map<char, string>& codes) {  unordered\_map<char, int> frequencyMap;  for (char ch : text) {  frequencyMap[ch]++;  }  // Приоритетная очередь для узлов с приоритетом наименьшей частоты  priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, CompareNodes> minHeap;  for (auto entry : frequencyMap) {  HuffmanNode\* node = new HuffmanNode(entry.first, entry.second);  minHeap.push(node);  }  // Строим дерево Хаффмана  while (minHeap.size() > 1) {  HuffmanNode\* left = minHeap.top();  minHeap.pop();  HuffmanNode\* right = minHeap.top();  minHeap.pop();  HuffmanNode\* mergedNode = new HuffmanNode('$', left->frequency + right->frequency);  mergedNode->left = left;  mergedNode->right = right;  minHeap.push(mergedNode);  }  // Получаем коды Хаффмана  HuffmanNode\* root = minHeap.top();  generateCodes(root, "", codes);  return root;  }  // Функция для кодирования текста с использованием кодов Хаффмана  string encodeText(string text, unordered\_map<char, string>& codes) {  string encodedText = "";  for (char ch : text) {  encodedText += codes[ch];  }  return encodedText;  }  // Функция для декодирования текста с использованием дерева Хаффмана  string decodeText(HuffmanNode\* root, string encodedText) {  string decodedText = "";  HuffmanNode\* current = root;  for (char bit : encodedText) {  if (bit == '0') {  current = current->left;  } else {  current = current->right;  }  if (!current->left && !current->right) {  decodedText += current->data;  current = root;  }  }  return decodedText;  }  // Основная функция для работы с алгоритмом Хаффмана  void huffmanMain(string text) {  unordered\_map<char, string> codes;  HuffmanNode\* root = buildHuffmanTree(text, codes);  // Закодировать текст  string encodedText = encodeText(text, codes);  // Расчет процента и коэффициента сжатия  double originalSize = text.size() \* 8.0; // каждый символ представлен 8 битами  double compressedSize = encodedText.size();  double compressionRatio = originalSize / compressedSize;  double compressionPercentage = (1 - (compressedSize / originalSize)) \* 100;  // Декодировать текст  string decodedText = decodeText(root, encodedText);  // Вывод результатов  cout << "Декодированный текст: " << decodedText << endl;  cout << "Закодированный текст: " << encodedText << endl;  cout << "Итоговый вес закодированного текста в битах: " << compressedSize << " бит" << endl;  cout << "Вес изначального текста в битах: " << originalSize << " бит" << endl;  cout << "Процент сжатия: " << compressionPercentage << "%" << endl;  cout << "Коэффициент сжатия: " << compressionRatio << endl;  }  int main(){  string filename = "text.txt";  ifstream inputFile(filename);  string fileContent((std::istreambuf\_iterator<char>(inputFile)), std::istreambuf\_iterator<char>());  inputFile.close();  string textContent = "черных тимофей юрьевич";  int mode;  cout << "Чтобы выполнить сжатие текстового файла алгоритмом Шеннона – Фано - нажмите 1" <<endl;  cout << "Чтобы выполнить сжатие строки {Черных Тимофей Юрьевич} алгоритмом Хаффмана - нажмите 2" <<endl;  cout << "Чтобы выполнить сжатие текстового файла алгоритмом Хаффмана - нажмите 3" <<endl;  cout << "Чтобы завершить работу - нажмите 0" <<endl;  cin >> mode;  while (mode != 0){  if (mode == 1){  ShannonFanoMain(fileContent);  }  else if (mode == 2){  huffmanMain(textContent);  }  else if (mode == 3){  huffmanMain(fileContent);  }  else{  break;  }  cout << "Чтобы выполнить сжатие текстового файла алгоритмом Шеннона – Фано - нажмите 1" <<endl;  cout << "Чтобы выполнить сжатие строки {Черных Тимофей Юрьевич} алгоритмом Хаффмана - нажмите 2" <<endl;  cout << "Чтобы выполнить сжатие текстового файла алгоритмом Хаффмана - нажмите 3" <<endl;  cout << "Чтобы завершить работу - нажмите 0" <<endl;  cin >> mode;  }  } |

Text.txt

|  |
| --- |
| В мире современных информационных технологий, алгоритмы сжатия текста играют ключевую роль в эффективном управлении и передаче данных. Эти алгоритмы предоставляют механизмы для уменьшения объема информации без потери существенных деталей, что существенно повышает эффективность хранения и передачи текстовых данных. Одним из пионеров в области алгоритмов сжатия текста является алгоритм Хаффмана, предложенный в 1952 году Дэвидом Хаффманом. Этот алгоритм основан на построении дерева, в котором символы с наименьшей частотой встречаемости находятся ближе к корню, что позволяет представить кодирование текста более эффективно. Другой замечательный алгоритм — LZ77, разработанный Абрахамом Лемпелем и Яаковом Зивем. Этот метод использует словарь и указатели на предыдущие вхождения, что позволяет эффективно представлять повторяющиеся фрагменты текста. Современные алгоритмы сжатия, такие как Deflate (используемый в форматах ZIP и PNG) или Brotli, представляют собой комбинации различных методов, включая словарное кодирование, адаптивные модели и прогрессивные техники сжатия. Алгоритмы сжатия текста не ограничиваются только локальными приложениями. Они находят широкое применение в передаче данных по сети, архивировании файлов, сжатии изображений и видео. Эти технологии являются основой для эффективной работы с данными в условиях ограниченной пропускной способности канала связи или хранилища. Таким образом, алгоритмы сжатия текста играют важную роль в обеспечении эффективности работы с данными, способствуя их компактному хранению и передаче, что является неотъемлемой частью современной информационной инфраструктуры. |

1. **Литература:**

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).