|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 7** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Кодирование и сжатие данных методами без потерь.»** | |
|  | |
| Выполнил студент группы ИКБО-03-21 | Хречко С.В. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2022

# **Цель работы**

Получение практических навыков и знаний по выполнению сжатия данных рассматриваемыми методами.

# **Постановка задачи**

Разработать программу в соответствии с требованиями варианта. Выполнить реализацию средствами ООП.

Вид дерева: дерево выражения.

Составить программу сжатия текста, используя метод RLE (run length encoding/кодирование длин серий/групповое кодирование). Рассчитать коэффициент сжатия в тестах.

Провести тестирование на длинной серии повторяющихся символов.

Провести тестирование на длинной серии неповторяющихся символов.

Составить программу сжатия текста алгоритмом Лемпеля-Зива (LZ77), LZ78. Рассчитать коэффициент сжатия в тестах.

Провести тестирование реализации LZ77 в соответствии с индивидуальным вариантом задания, используя двухсимвольный алфавит (0, 1). Описать в отчете процесс восстановления полученного сжатого текста.

Провести тестирование реализации LZ78 в соответствии с индивидуальным вариантом задания. Описать в отчете процесс восстановления полученного сжатого текста.

Составить программу сжатия и восстановления текста алгоритмами Шеннона-Фано и Хаффмана.

Провести тестирование метода Шеннона-Фано в соответствии с индивидуальным вариантом задания. Представить в отчете таблицу формирования кода, изобразить префиксное дерево, рассчитать коэффициент сжатия.

Провести тестирование метода Хаффмана на строке, содержащей ваше ФИО. Построить таблицу частот встречаемости символов, сформировать алфавит исходной строки и посчитать частоту вхождений и вероятность появления символов. Изобразить префиксное дерево Хаффмана. Показать кодирование каждого символа в строке. Рассчитать коэффициент сжатия, среднюю длину кода и дисперсию.

Применить алгоритм к большому текстовому файлу. Архивировать тот же файл любым архиватором. Выполнить сравнительный анализ сжатия этими способами.

Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Вариант №9. Условие задания:

|  |  |  |
| --- | --- | --- |
| 000100101100100010001 | kloklonkolonklonkl | Эни-бени рити-Фати. Дорба, дорба сентибрати.  Дэл. Дэл. Кошка. Дэл. Фати! |

# **Решение**

Сжа́тие да́нных — алгоритмическое (обычно обратимое) преобразование данных, производимое с целью уменьшения занимаемого ими объёма. Применяется для более рационального использования устройств хранения и передачи данных.

Сжатие основано на устранении избыточности, содержащейся в исходных данных. Простейшим примером избыточности является повторение в тексте фрагментов (например, слов естественного или машинного языка). Подобная избыточность обычно устраняется заменой повторяющейся последовательности ссылкой на уже закодированный фрагмент с указанием его длины. Другой вид избыточности связан с тем, что некоторые значения в сжимаемых данных встречаются чаще других. Сокращение объёма данных достигается за счёт замены часто встречающихся данных короткими кодовыми словами, а редких — длинными (энтропийное кодирование). Сжатие данных, не обладающих свойством избыточности (например, случайный сигнал или белый шум, зашифрованные сообщения), принципиально невозможно без потерь.

По заданию требовалось реализовать rle сжатие. Функция проходится по переданной строке, подсчитывает количество повторяющихся символов и заменяет повторения количеством повторяющихся символов.

|  |
| --- |
| string rleCompression(string s) {      string result = "";      int count = 1;      for (int i = 0; i < s.length(); i++) {          if (s[i] == s[i + 1]) {              count++;          } else {              if(count < 5){                  for (int j = 0; j < count; j++) {                      result += s[i];                  }              } else {                  result += s[i];                  result += "<" + to\_string(count) + ">";              }              count = 1;          }      }      return result;  } |

Для проверки корректности работы rle сжатия была создана функция восстановления данных. Функция находит записи о повторяющихся символах и выводит в выходную строку требуемое символов.

|  |
| --- |
| string rleDeCompression(string s) {      string result = "";      int i = 0;      while (i < s.length()) {          if (s[i] == '<') {              int j = i + 1;              int count = stoi(s.substr(j, s.find('>', j) - j));              for (int k = 0; k < count-1; k++) {                  result += s[i - 1];              }              i = s.find('>', j) + 1;          } else {              result += s[i];              i++;          }      }      return result;  } |

По условию задания требования требовалось реализовать сжатие по алгоритму lz77. Функция заменяет встречающиеся ранее в определенном буфере наборы символов, на отметки о сдвиге назад по строке, а также количестве символов, которые после данного сдвига нужно скопировать, чтобы восстановить запись.

|  |
| --- |
| struct NodeLZ77{      int offset;      int length;      char next;  };  vector<NodeLZ77> lz77Compression(string s, int windowSize, int lookAheadSize) {      vector<NodeLZ77> result;      int i = 0;      while (i < s.length()) {          int offset = 0;          int length = 0;          int max = findMatch(s, i, windowSize, lookAheadSize, offset, length);          if (max == 0) {              NodeLZ77 node;              node.offset = 0;              node.length = 0;              node.next = s[i];              result.push\_back(node);              i++;          } else {              NodeLZ77 node;              node.offset = offset;              node.length = length;              node.next = s[i + length];              result.push\_back(node);              i += length + 1;          }      }      return result;  } |

Для функционирования lz77 сжатия требуется функция поиска совпадений в буфере. Функция проходится по строке на переданное количество символов назад и ищет совпадения с буфером, размер которого также передается в функцию.

|  |
| --- |
| int findMatch(string s, int i, int windowSize, int lookAheadSize, int &offset, int &length) {      int max = 0;      for (int j = i - windowSize; j < i; j++) {          if (j < 0) {              continue;          }          int k = 0;          while (s[i + k] == s[j + k] && k < lookAheadSize) {              k++;          }          if (k > max) {              max = k;              offset = i - j;              length = k;          }      }      return max;  } |

Для проверки корректности работы lz77 сжатия была создана функция восстановления данных. Функция находит записи, говорящие о повторяющихся последовательностях символов, возвращается на указанное число символов назад и копирует в выходную строку предписанное количество символов.

|  |
| --- |
| string lz77DeCompression(vector<NodeLZ77> s) {      string result = "";      for (int i = 0; i < s.size(); i++) {          if (s[i].length == 0) {              result += s[i].next;          } else {              int start = result.length() - s[i].offset;              for (int j = 0; j < s[i].length; j++) {                  result += result[start + j];              }              result += s[i].next;          }      }      return result;  } |

По условию задания требования требовалось реализовать сжатие по алгоритму lz78. Функция сохраняет элементы строки в словарь, чтобы в случае повторного нахождения таких элементов заменить их на ссылку, на соответствующий элемент словаря.

|  |
| --- |
| struct LZ78Node {      int index;      char next;  };  vector<LZ78Node> lz78Compression(string s) {      vector<LZ78Node> result;      map<string, int> dictionary;      string temp = "";      for (int i = 0; i < s.length(); i++) {          if(dictionary.find(temp + s[i]) != dictionary.end()){              temp += s[i];          }          else{              LZ78Node node;              node.index = dictionary[temp];              node.next = s[i];              result.push\_back(node);              dictionary[temp + s[i]] = dictionary.size();              temp = "";          }      }      if(temp != ""){          LZ78Node node;          node.next = temp[temp.length() - 1];          temp = temp.substr(0, temp.length() - 1);          node.index = dictionary[temp];          result.push\_back(node);      }      return result;  } |

Для проверки корректности работы lz78 сжатия была создана функция восстановления данных. Функция строит словарь аналогично функции сжатия. При нахождении ссылки на словарь, функция подставляет соответствующую строку из словаря.

|  |
| --- |
| string lz78DeCompression(vector<LZ78Node> s) {      string result = "";      vector<string> dictionary = {""};      for (int i = 0; i < s.size(); i++) {          string temp = dictionary[s[i].index] + s[i].next;          result += temp;          dictionary.push\_back(temp);      }      return result;  } |

По условию задачи требовалось разработать функцию, выполняющую сжатие метод Шеннона-Фано. Основная функция заменяет символы, на код, который до этого создается другой функцией.

|  |
| --- |
| string shennonFanoCompression(map<char, string> code, string s) {      string result = "";      for (int i = 0; i < s.length(); i++) {          result += code[s[i]];      }      return result;  } |

Для работы функции сжатия методом Шеннона-Фано требуется функция создающая коды символов. Функция принимает количество вхождений различных символов, которое подсчитывает отдельная функция. Функция разделяет символы на две части так, чтобы каждая часть имело примерно равное суммарное количество вхождений в строку. Каждой из двух частей дописывается 0 или 1 в кодирующее слово. Далее функция выполняет данные действия рекурсивно, пока не получит кодирующие слова для каждого отдельного символа.

|  |
| --- |
| map<char, string> shennonFanoCompressionMap(const map<char, int>& count) {      map<char, string> result;      vector<pair<char, int>> temp;      for (auto it = count.begin(); it != count.end(); it++) {          temp.push\_back(\*it);      }      sort(temp.begin(), temp.end(), [](pair<char, int> a, pair<char, int> b) {return a.second > b.second; });      int sum = 0;      for (int i = 0; i < temp.size(); i++) {          sum += temp[i].second;      }      vector<pair<char, int>> left;      vector<pair<char, int>> right;      int sum1 = 0;      int sum2 = 0;      for(int i = 0; i < temp.size(); i++){          if(sum1 <= sum2){              left.push\_back(temp[i]);              sum1 += temp[i].second;          }          else{              right.push\_back(temp[i]);              sum2 += temp[i].second;          }      }      for (int k = 0; k < left.size(); k++) {          result[left[k].first] = "0";      }      for (int k = 0; k < right.size(); k++) {          result[right[k].first] = "1";      }      //now we go recursive for left and right      if (left.size() > 1) {          map<char, int> tempCount;          for (int i = 0; i < left.size(); i++) {              tempCount[left[i].first] = left[i].second;          }          map<char, string> tempLeaf = shennonFanoCompressionMap(tempCount);          for (auto it = tempLeaf.begin(); it != tempLeaf.end(); it++) {              result[it->first] = "0" + it->second;          }      }      if (right.size() > 1) {          map<char, int> tempCount;          for (int i = 0; i < right.size(); i++) {              tempCount[right[i].first] = right[i].second;          }          map<char, string> tempLeaf = shennonFanoCompressionMap(tempCount);          for (auto it = tempLeaf.begin(); it != tempLeaf.end(); it++) {              result[it->first] = "1" + it->second;          }      }      return result;  } |

Для проверки корректности работы алгоритма Шеннона-Фано была создана функция восстановления данных. Функция принимает одним из параметров таблицу кодов символов. Функция проходит по строке, находя коды символов и выводя в выходную строку соответствующий коду символ.

|  |
| --- |
| string shennonFanoDeCompression(map<char, string> code, string s) {      string result = "";      string temp = "";      for (int i = 0; i < s.length(); i++) {          temp += s[i];          for (auto it = code.begin(); it != code.end(); it++) {              if (it->second == temp) {                  result += it->first;                  temp = "";                  break;              }          }      }      return result;  } |

Таблица генерации кодов для задания представленного в варианте по коду Шеннона-Фано изображена на рисунке 1.

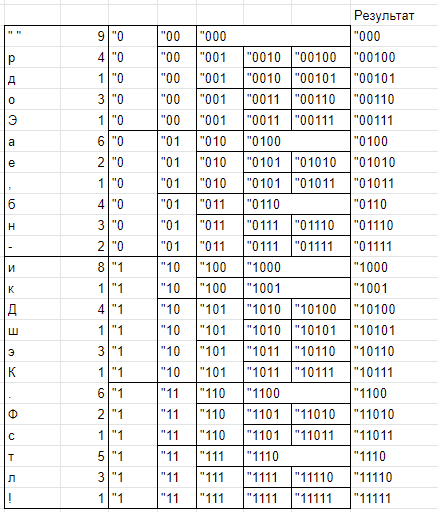


Рисунок 1. Таблица генерации кодов

Префиксное дерево по этим кодам представлено на рисунках 2 и 3.

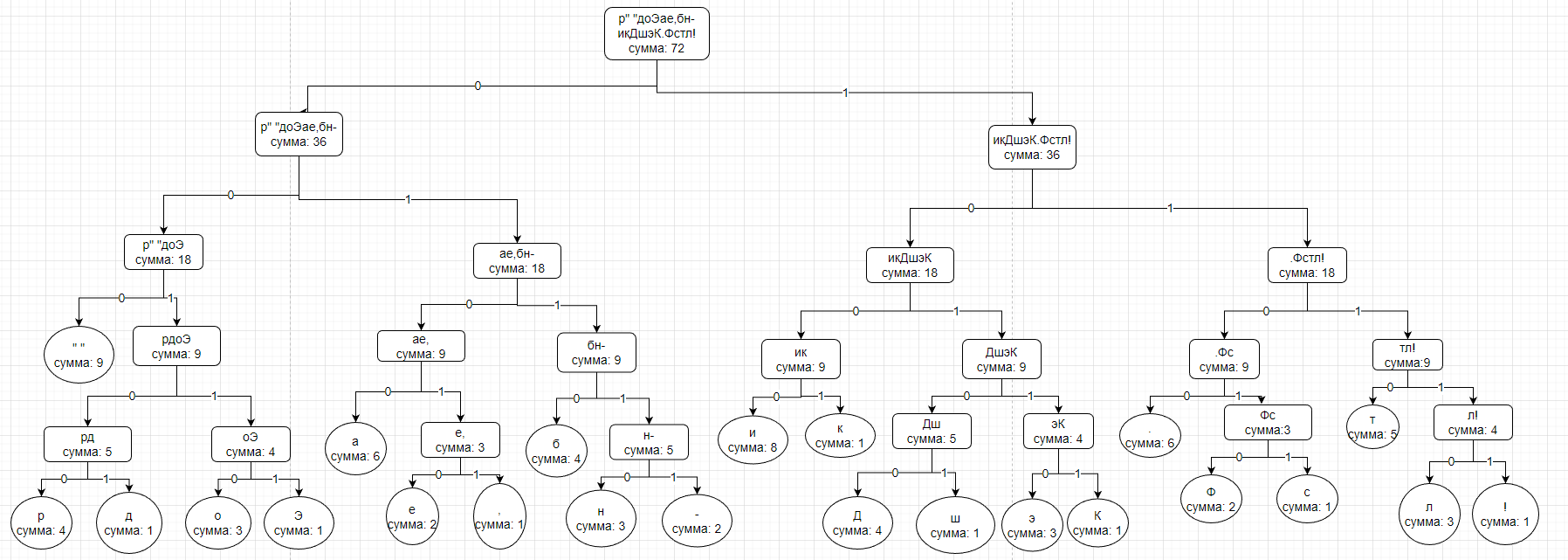


Рисунок 2. Префиксное дерево

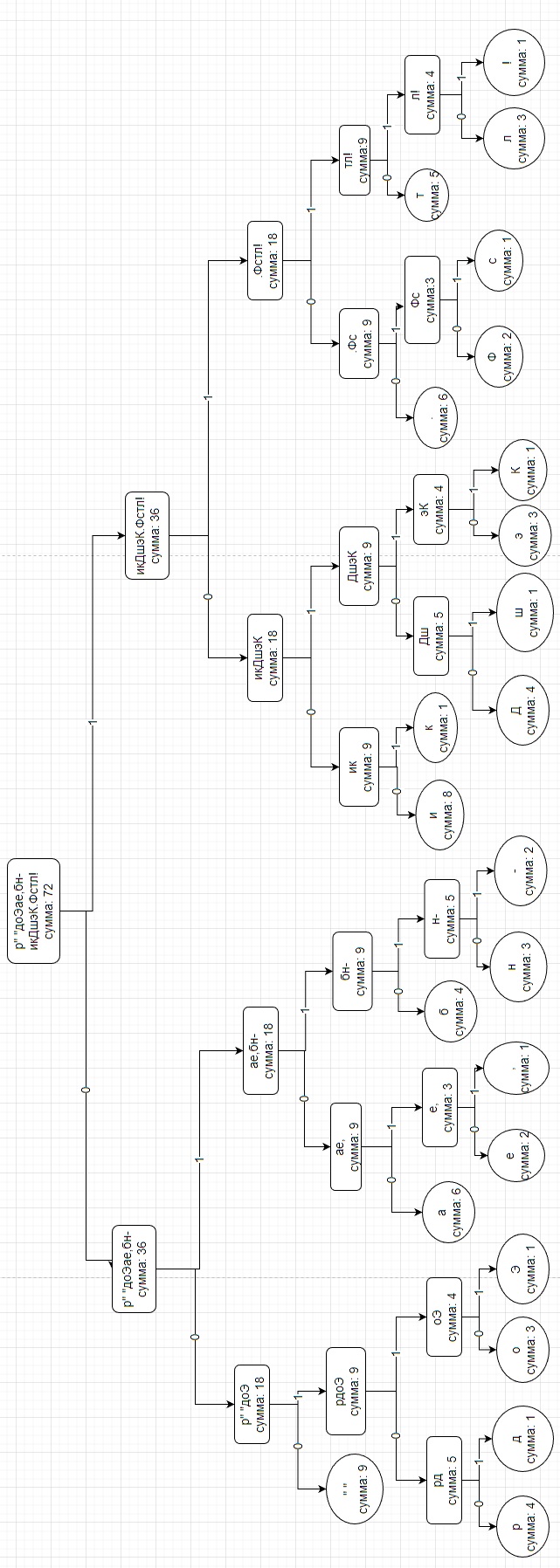


Рисунок 3. Префиксное дерево

По условию задачи требовалось разработать функцию, выполняющую сжатие методом Хаффмана. Основная функция заменяет символы, на код, который до этого создается другой функцией.

|  |
| --- |
| string shennonFanoCompression(map<char, string> code, string s) {      string result = "";      for (int i = 0; i < s.length(); i++) {          result += code[s[i]];      }      return result;  } |

Для функционирования метода Хаффмана используется бинарное дерево, узел которого представлен далее.

|  |
| --- |
| struct HaffmanBinaryTreeNode {      char c;      int count;      HaffmanBinaryTreeNode\* left;      HaffmanBinaryTreeNode\* right;      HaffmanBinaryTreeNode(char c, int count) {          this->c = c;          this->count = count;          this->left = nullptr;          this->right = nullptr;      }      HaffmanBinaryTreeNode(HaffmanBinaryTreeNode\* left, HaffmanBinaryTreeNode\* right) {          this->c = 0;          this->count = left->count + right->count;          this->left = left;          this->right = right;      }      ~HaffmanBinaryTreeNode() {          if (this->left != nullptr) {              delete this->left;          }          if (this->right != nullptr) {              delete this->right;          }      }  }; |

Для функционирования сжатия Хаффмана требуется функция создающая коды для символов. Функция принимает количество вхождений символов в строку, которое подсчитывается функцией, которая была рассмотрена ранее. Функция создает бинарное дерево по алгоритму Хаффмана, после чего вызывает другую функцию, которая рекурсивно строит коды для символов.

|  |
| --- |
| map<char, string> haffmanCompressionMap(const map<char, int>& count) {      map<char, string> result;      vector<HaffmanBinaryTreeNode\*> temp;      for (auto it = count.begin(); it != count.end(); it++) {          temp.push\_back(new HaffmanBinaryTreeNode(it->first, it->second));      }      while (temp.size() > 1) {          sort(temp.begin(), temp.end(), [](HaffmanBinaryTreeNode\* a, HaffmanBinaryTreeNode\* b) {return a->count > b->count; }); //haha press f to pay respect to efficiency          HaffmanBinaryTreeNode\* left = temp.back();          temp.pop\_back();          HaffmanBinaryTreeNode\* right = temp.back();          temp.pop\_back();          temp.push\_back(new HaffmanBinaryTreeNode(left, right));      }      HaffmanBinaryTreeNode\* root = temp.back();      temp.pop\_back();      //now we have haffman tree      //we go recursive and fill map      haffmanCompressionMapRecursive(root, result);      delete root;      return result;  } |

Для функционирования сжатия Хаффмана требуется функция, которая будет создавать коды символов по созданному бинарному дереву. Функция идет рекурсивно по дереву и записывает нули или единицы в строку кода, основываясь на том по левому поддереву пошла функция или по правому. При нахождении листа дерева, функция присваивает сохраненному там символу полученный код.

|  |
| --- |
| void haffmanCompressionMapRecursive(HaffmanBinaryTreeNode\* node, map<char, string>& result, string code = "") {      if (node->left == nullptr && node->right == nullptr) {          result[node->c] = code;          return;      }      haffmanCompressionMapRecursive(node->left, result, code + "0");      haffmanCompressionMapRecursive(node->right, result, code + "1");  } |

Для проверки работоспособности была создана функция восстановления данных. Функция заменяет встреченные коды символов, на соответствующие им символы.

|  |
| --- |
| string haffmanDeCompression(map<char, string> code, string s) {      string result = "";      string temp = "";      for (int i = 0; i < s.length(); i++) {          temp += s[i];          for (auto it = code.begin(); it != code.end(); it++) {              if (it->second == temp) {                  result += it->first;                  temp = "";                  break;              }          }      }      return result;  } |

Частота встречаемости, вероятность появления в строке символов для задания по кодированию Хаффмана (ФИО) представлены на рисунке 4.

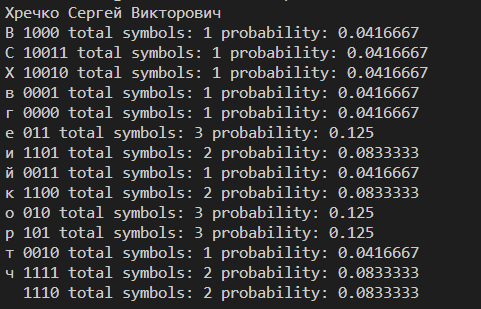


Рисунок 4. Встречаемости символов

Сформированные кодовые слова представлены на рисунке 5.

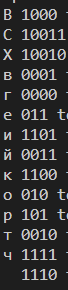


Рисунок 5. Коды символов

Средняя длина кодов и дисперсия представлены на рисунке 6.



Рисунок 6. Средняя длина и дисперсия

Префиксное дерево по этим кодам представлено на рисунке 7.

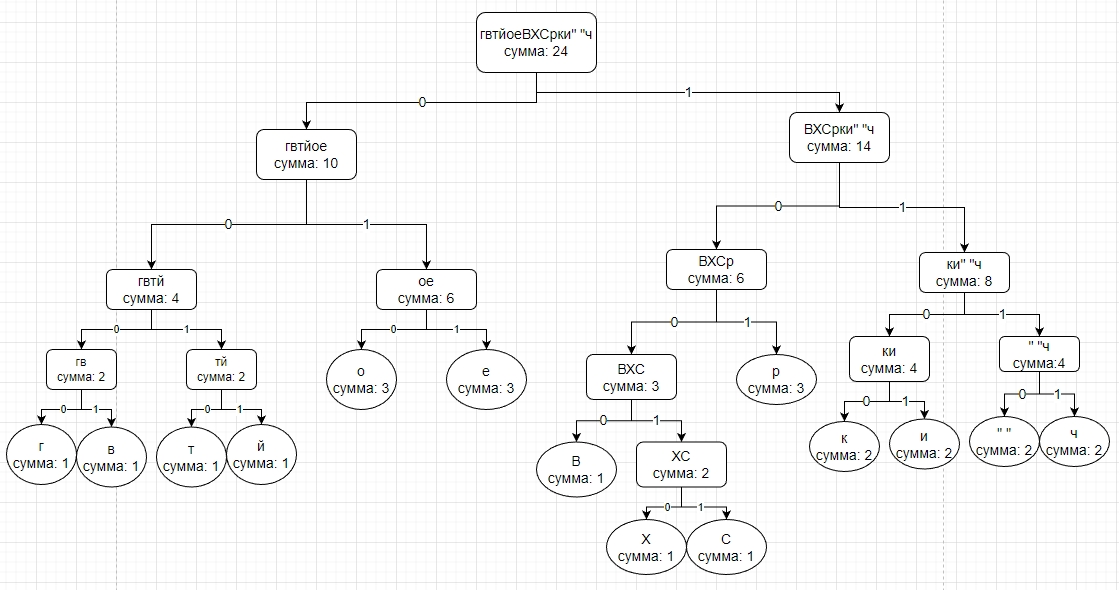


Рисунок 7. Префиксное дерево

Пользовательский интерфейс программы предлагает пользователю вводить строки для сжатия их различными методами сжатия, после чего выводит сжатые данные, а также результаты восстановления данных и исходные строки для тестирования корректности работы алгоритмов, пользовательский интерфейс представлен на рисунке 8.

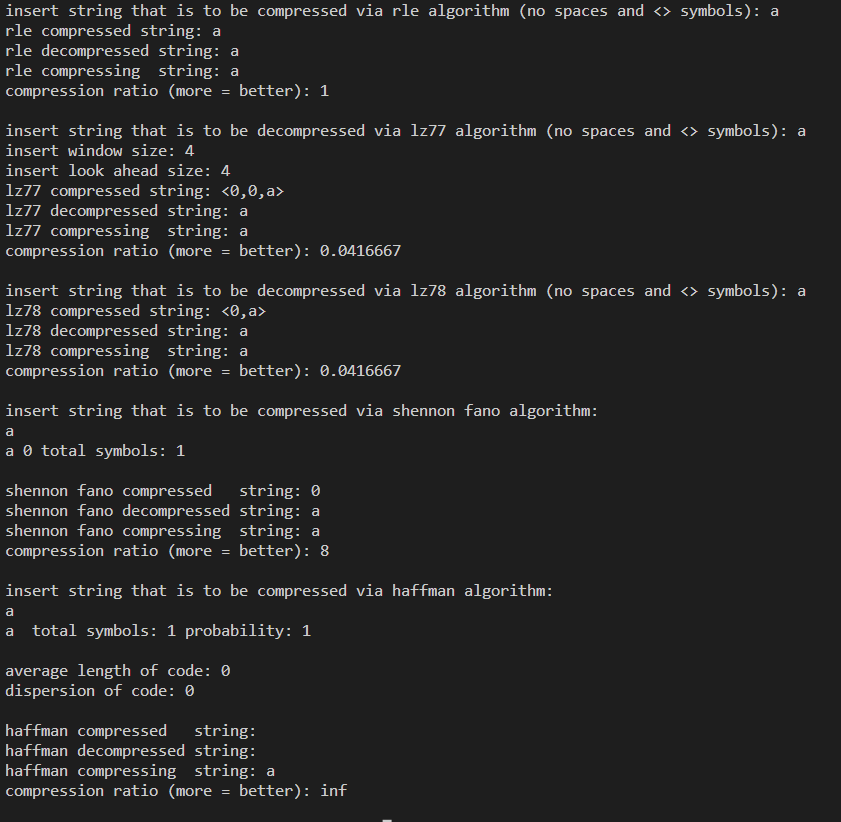


Рисунок 8. Интерфейс программы

# **Тестирование**

Тестирование rle сжатия. На рисунке 9 представлено тестирование на длинной последовательности повторяющихся символов, а на рисунке 10 – неповторяющихся. Как мы можем видеть, на неповторяющихся символах алгоритм показывает крайне низкую эффективность.

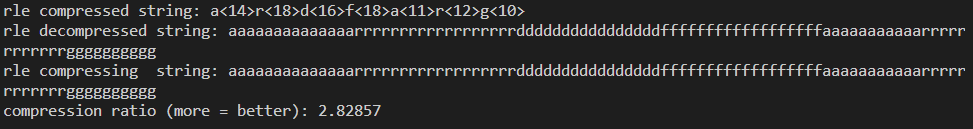


Рисунок 9. Тестирование rle сжатия

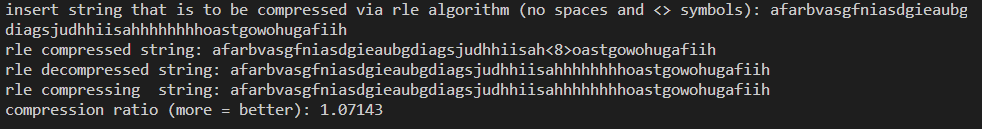


Рисунок 10. Тестирование rle сжатия

Тестирование lz77 сжатия по заданию из варианта представлено на рисунке 11. Мы можем видеть, что функция восстановления данных, верно выполнила свою работу, а значит и сжатие было выполнено верно.

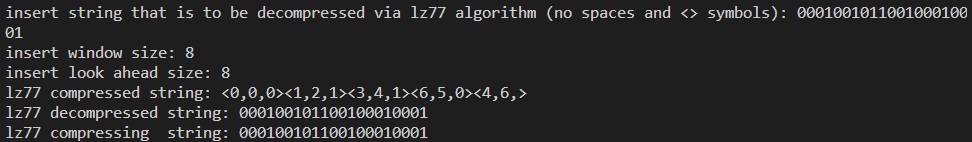


Рисунок 11. Тестирование lz77 сжатия

Тестирование lz78 сжатия по заданию из варианта представлено на рисунке 12. Мы можем видеть, что функция восстановления данных, верно выполнила свою работу, а значит и сжатие было выполнено верно.

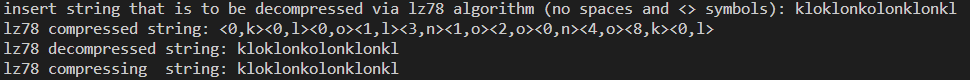


Рисунок 12. Тестирование lz78 сжатия

Тестирование сжатия методом Шеннона-Фано по заданию из варианта представлено на рисунке 13. Мы можем видеть, что сжатия и восстановление были выполнены успешно. Также на рисунке представлен коэффициент сжатия.

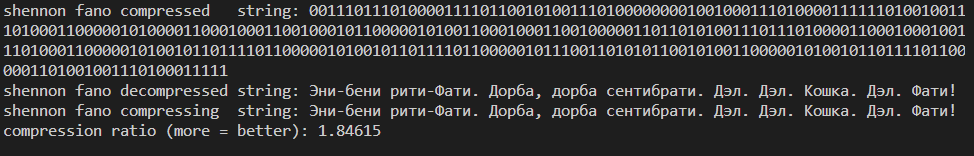


Рисунок 13. Тестирование Шеннона-Фано

Тестирование сжатия методом Хаффмана в соответствии с задание на строке представляющей ФИО студента представлено на рисунке 14. Мы можем видеть, что сжатия и восстановление были выполнены успешно. Также на рисунке представлен коэффициент сжатия.

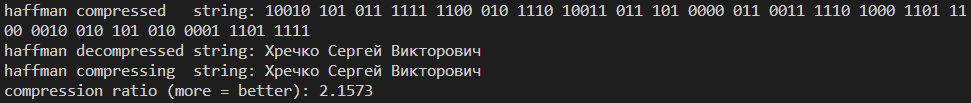


Рисунок 13. Тестирование Хаффмана

Сравнительное тестирование написанных алгоритмов Шеннона-Фано и Хаффмана с архиватором.

Compression ratio – это отношение веса не сжатого текста к весу сжатого. Текст для тестирования был взят со следующего сайта <https://habr.com/ru/post/144200/>.

Результаты написанных алгоритмов Шеннона-Фано и Хаффмана представлены на рисунках 14 и 15 соответственно.



Рисунок 14. Коэффициент сжатия для Шеннона-Фано



Рисунок 15. Коэффициент сжатия для Хаффмана

Теперь протестируем сжатие программой WinRAR archiver. Изначальный размер файла представлен на рисунке 16.

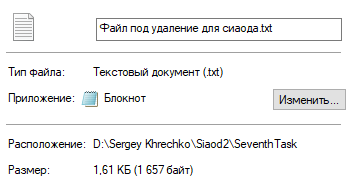


Рисунок 16. Изначальный размер файла

Размер файла после сжатия представлен на рисунке 17.

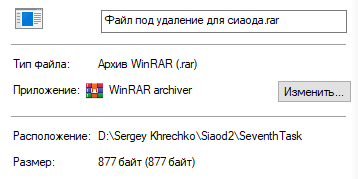


Рисунок 17. Размер файла после сжатия

Воспользовавшись обширными познаниями в математике и используя возможности современных вычислительных машин, находим коэффициент сжатия и округляем его до 5-ти знаков после запятой. Коэффициент сжатия равен 1,88940. Из тестирования видно, что WinRAR archiver показывает лучшие результаты, чем два написанных нами алгоритма.

Из результатов выполнения программы видно:

1. Программа работает корректно, все функции работоспособны.

# **Вывод**

В результате выполнения работы я:

1. Освоил базовую работу алгоритмов сжатия данных.
2. Научился разрабатывать и тестировать алгоритмы сжатия данных.

# **Исходный код программы**

|  |
| --- |
| #include <iostream>  #include <string>  #include <map>  #include <vector>  #include <algorithm>  #include <queue>  #include <functional>  using namespace std;  string rleCompression(string s) {      string result = "";      int count = 1;      for (int i = 0; i < s.length(); i++) {          if (s[i] == s[i + 1]) {              count++;          } else {              if(count < 5){                  for (int j = 0; j < count; j++) {                      result += s[i];                  }              } else {                  result += s[i];                  result += "<" + to\_string(count) + ">";              }              count = 1;          }      }      return result;  }  string rleDeCompression(string s) {      string result = "";      int i = 0;      while (i < s.length()) {          if (s[i] == '<') {              int j = i + 1;              int count = stoi(s.substr(j, s.find('>', j) - j));              for (int k = 0; k < count-1; k++) {                  result += s[i - 1];              }              i = s.find('>', j) + 1;          } else {              result += s[i];              i++;          }      }      return result;  }  int findMatch(string s, int i, int windowSize, int lookAheadSize, int &offset, int &length) {      int max = 0;      for (int j = i - windowSize; j < i; j++) {          if (j < 0) {              continue;          }          int k = 0;          while (s[i + k] == s[j + k] && k < lookAheadSize) {              k++;          }          if (k > max) {              max = k;              offset = i - j;              length = k;          }      }      return max;  }  struct NodeLZ77{      int offset;      int length;      char next;  };  vector<NodeLZ77> lz77Compression(string s, int windowSize, int lookAheadSize) {      vector<NodeLZ77> result;      int i = 0;      while (i < s.length()) {          int offset = 0;          int length = 0;          int max = findMatch(s, i, windowSize, lookAheadSize, offset, length);          if (max == 0) {              NodeLZ77 node;              node.offset = 0;              node.length = 0;              node.next = s[i];              result.push\_back(node);              i++;          } else {              NodeLZ77 node;              node.offset = offset;              node.length = length;              node.next = s[i + length];              result.push\_back(node);              i += length + 1;          }      }      return result;  }  string lz77DeCompression(vector<NodeLZ77> s) {      string result = "";      for (int i = 0; i < s.size(); i++) {          if (s[i].length == 0) {              result += s[i].next;          } else {              int start = result.length() - s[i].offset;              for (int j = 0; j < s[i].length; j++) {                  result += result[start + j];              }              result += s[i].next;          }      }      return result;  }  struct LZ78Node {      int index;      char next;  };  vector<LZ78Node> lz78Compression(string s) {      vector<LZ78Node> result;      map<string, int> dictionary;      string temp = "";      for (int i = 0; i < s.length(); i++) {          if(dictionary.find(temp + s[i]) != dictionary.end()){              temp += s[i];          }          else{              LZ78Node node;              node.index = dictionary[temp];              node.next = s[i];              result.push\_back(node);              dictionary[temp + s[i]] = dictionary.size();              temp = "";          }      }      if(temp != ""){          LZ78Node node;          node.next = temp[temp.length() - 1];          temp = temp.substr(0, temp.length() - 1);          node.index = dictionary[temp];          result.push\_back(node);      }      return result;  }  string lz78DeCompression(vector<LZ78Node> s) {      string result = "";      vector<string> dictionary = {""};      for (int i = 0; i < s.size(); i++) {          string temp = dictionary[s[i].index] + s[i].next;          result += temp;          dictionary.push\_back(temp);      }      return result;  }  map<char, int> countCharacters(string s) {      map<char, int> result;      for (int i = 0; i < s.length(); i++) {          if (result.find(s[i]) == result.end()) {              result[s[i]] = 1;          } else {              result[s[i]]++;          }      }      return result;  }  map<char, string> shennonFanoCompressionMap(const map<char, int>& count) {      map<char, string> result;      vector<pair<char, int>> temp;      for (auto it = count.begin(); it != count.end(); it++) {          temp.push\_back(\*it);      }      sort(temp.begin(), temp.end(), [](pair<char, int> a, pair<char, int> b) {return a.second > b.second; });      int sum = 0;      for (int i = 0; i < temp.size(); i++) {          sum += temp[i].second;      }      vector<pair<char, int>> left;      vector<pair<char, int>> right;      int sum1 = 0;      int sum2 = 0;      for(int i = 0; i < temp.size(); i++){          if(sum1 <= sum2){              left.push\_back(temp[i]);              sum1 += temp[i].second;          }          else{              right.push\_back(temp[i]);              sum2 += temp[i].second;          }      }      for (int k = 0; k < left.size(); k++) {          result[left[k].first] = "0";      }      for (int k = 0; k < right.size(); k++) {          result[right[k].first] = "1";      }      //now we go recursive for left and right      if (left.size() > 1) {          map<char, int> tempCount;          for (int i = 0; i < left.size(); i++) {              tempCount[left[i].first] = left[i].second;          }          map<char, string> tempLeaf = shennonFanoCompressionMap(tempCount);          for (auto it = tempLeaf.begin(); it != tempLeaf.end(); it++) {              result[it->first] = "0" + it->second;          }      }      if (right.size() > 1) {          map<char, int> tempCount;          for (int i = 0; i < right.size(); i++) {              tempCount[right[i].first] = right[i].second;          }          map<char, string> tempLeaf = shennonFanoCompressionMap(tempCount);          for (auto it = tempLeaf.begin(); it != tempLeaf.end(); it++) {              result[it->first] = "1" + it->second;          }      }      return result;  }  string shennonFanoCompression(map<char, string> code, string s) {      string result = "";      for (int i = 0; i < s.length(); i++) {          result += code[s[i]];      }      return result;  }  string shennonFanoDeCompression(map<char, string> code, string s) {      string result = "";      string temp = "";      for (int i = 0; i < s.length(); i++) {          temp += s[i];          for (auto it = code.begin(); it != code.end(); it++) {              if (it->second == temp) {                  result += it->first;                  temp = "";                  break;              }          }      }      return result;  }  struct HaffmanBinaryTreeNode {      char c;      int count;      HaffmanBinaryTreeNode\* left;      HaffmanBinaryTreeNode\* right;      HaffmanBinaryTreeNode(char c, int count) {          this->c = c;          this->count = count;          this->left = nullptr;          this->right = nullptr;      }      HaffmanBinaryTreeNode(HaffmanBinaryTreeNode\* left, HaffmanBinaryTreeNode\* right) {          this->c = 0;          this->count = left->count + right->count;          this->left = left;          this->right = right;      }      ~HaffmanBinaryTreeNode() {          if (this->left != nullptr) {              delete this->left;          }          if (this->right != nullptr) {              delete this->right;          }      }  };  void haffmanCompressionMapRecursive(HaffmanBinaryTreeNode\* node, map<char, string>& result, string code = "") {      if (node->left == nullptr && node->right == nullptr) {          result[node->c] = code;          return;      }      haffmanCompressionMapRecursive(node->left, result, code + "0");      haffmanCompressionMapRecursive(node->right, result, code + "1");  }  map<char, string> haffmanCompressionMap(const map<char, int>& count) {      map<char, string> result;      vector<HaffmanBinaryTreeNode\*> temp;      for (auto it = count.begin(); it != count.end(); it++) {          temp.push\_back(new HaffmanBinaryTreeNode(it->first, it->second));      }      while (temp.size() > 1) {          sort(temp.begin(), temp.end(), [](HaffmanBinaryTreeNode\* a, HaffmanBinaryTreeNode\* b) {return a->count > b->count; }); //haha press f to pay respect to efficiency          HaffmanBinaryTreeNode\* left = temp.back();          temp.pop\_back();          HaffmanBinaryTreeNode\* right = temp.back();          temp.pop\_back();          temp.push\_back(new HaffmanBinaryTreeNode(left, right));      }      HaffmanBinaryTreeNode\* root = temp.back();      temp.pop\_back();      //now we have haffman tree      //we go recursive and fill map      haffmanCompressionMapRecursive(root, result);      delete root;      return result;  }  string haffmanCompression(map<char, string> code, string s) {      string result = "";      for (int i = 0; i < s.length(); i++) {          result += code[s[i]] + " ";      }      return result;  }  string haffmanDeCompression(map<char, string> code, string s) {      string result = "";      string temp = "";      for (int i = 0; i < s.length(); i++) {          if(s[i] == ' ') continue;          temp += s[i];          for (auto it = code.begin(); it != code.end(); it++) {              if (it->second == temp) {                  result += it->first;                  temp = "";                  break;              }          }      }      return result;  }    int main() {      cout << "insert string that is to be compressed via rle algorithm (no spaces and <> symbols): ";      string s;      cin >> s;      cin.clear();      cin.ignore(10000, '\n');      string compressed = rleCompression(s);      cout << "rle compressed string: " << compressed << endl;      string decompressed = rleDeCompression(compressed);      cout << "rle decompressed string: " << decompressed << endl;      cout << "rle compressing  string: " << s << endl;      cout << "compression ratio (more = better): " << (double)(s.size() \* sizeof(char)) / (double)(compressed.size() \* sizeof(char)) << endl;      cout << endl;      cout << "insert string that is to be decompressed via lz77 algorithm (no spaces and <> symbols): ";      cin >> s;      cin.clear();      cin.ignore(10000, '\n');      cout << "insert window size: ";      int windowSize;      cin >> windowSize;      cin.clear();      cin.ignore(10000, '\n');      cout << "insert look ahead size: ";      int lookAheadSize;      cin >> lookAheadSize;      cin.clear();      cin.ignore(10000, '\n');      vector<NodeLZ77> compressedVec = lz77Compression(s, windowSize, lookAheadSize);      cout << "lz77 compressed string: ";      for (int i = 0; i < compressedVec.size(); i++) {          cout << "<" << compressedVec[i].offset << "," << compressedVec[i].length << "," << compressedVec[i].next << ">";      }      cout << endl;      decompressed = lz77DeCompression(compressedVec);      cout << "lz77 decompressed string: " << decompressed << endl;      cout << "lz77 compressing  string: " << s << endl;      cout << "compression ratio (more = better): " << (double)(s.size()\*sizeof(char)) / (double)sizeof(compressedVec) << endl;      cout << endl;      cout << "insert string that is to be decompressed via lz78 algorithm (no spaces and <> symbols): ";      cin >> s;      cin.clear();      cin.ignore(10000, '\n');      vector<LZ78Node> compressedVec2 = lz78Compression(s);      cout << "lz78 compressed string: ";      for (int i = 0; i < compressedVec2.size(); i++) {          cout << "<" << compressedVec2[i].index << "," << compressedVec2[i].next << ">";      }      cout << endl;      decompressed = lz78DeCompression(compressedVec2);      cout << "lz78 decompressed string: " << decompressed << endl;      cout << "lz78 compressing  string: " << s << endl;      cout << "compression ratio (more = better): " << (double)(s.size()\*sizeof(char)) / (double)sizeof(compressedVec2) << endl;      cout << endl;      cout << "insert string that is to be compressed via shennon fano algorithm: " << endl;      getline(cin, s);      //s = rleDeCompression("a<50>b<39>c<18>d<49>e<35>f<24>"); //test from wiki        map<char, int> count = countCharacters(s);      map<char, string> compressedMap = shennonFanoCompressionMap(count);      //print the compressed map      for (auto it = compressedMap.begin(); it != compressedMap.end(); it++) {          cout << it->first << " " << it->second;          cout << " total symbols: " << count[it->first] << endl;      }      cout << endl;      compressed = shennonFanoCompression(compressedMap, s);      cout << "shennon fano compressed   string: " << compressed << endl;      decompressed = shennonFanoDeCompression(compressedMap, compressed);      cout << "shennon fano decompressed string: " << decompressed << endl;      cout << "shennon fano compressing  string: " << s << endl;      //here we compare the string as a sequence of chars (every char is 1 byte)      //with the compressed string as a sequence of bits (every bit is 1/8 byte)      //that's why s is multiplyed by 8 and compressed by 1      cout << "compression ratio (more = better): " << (double)(s.size()\*8) / (double)(compressed.size()) << endl;      cout << endl;      cout << "insert string that is to be compressed via haffman algorithm: " << endl;      getline(cin, s);      count = countCharacters(s);      compressedMap = haffmanCompressionMap(count);      //print the compressed map      for (auto it = compressedMap.begin(); it != compressedMap.end(); it++) {          cout << it->first << " " << it->second;          cout << " total symbols: " << count[it->first] << " probability: " << (double)count[it->first] / (double)s.size() << endl;      }      cout << endl;      cout << "average length of code: ";      double averageLength = 0;      for (auto it = compressedMap.begin(); it != compressedMap.end(); it++) {          averageLength += it->second.size();      }      averageLength /= compressedMap.size();      cout << averageLength << endl;      cout << "dispersion of code: ";      double dispersion = 0;      for (auto it = compressedMap.begin(); it != compressedMap.end(); it++) {          dispersion += (it->second.size() - averageLength) \* (it->second.size() - averageLength);      }      dispersion /= compressedMap.size();      cout << dispersion << endl;      cout << endl;      compressed = haffmanCompression(compressedMap, s);      cout << "haffman compressed   string: " << compressed << endl;      decompressed = haffmanDeCompression(compressedMap, compressed);      cout << "haffman decompressed string: " << decompressed << endl;      cout << "haffman compressing  string: " << s << endl;      int spaces = 0;      for (int i = 0; i < compressed.size(); i++) {          if (compressed[i] == ' ') {              spaces++;          }      }      cout << "compression ratio (more = better): " << (double)(s.size() \* 8) / (double)(compressed.size() - spaces) << endl;      cout << endl;      return 0;  } |