|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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;  } |

Пользовательский интерфейс программы предлагает пользователю ввести выражение. Затем выводит информацию по этому выражению: построенное дерево, является ли выражение алгебраическим, и затем выводит восстановленное выражение и проверяет присутствует ли в нем символ \*.

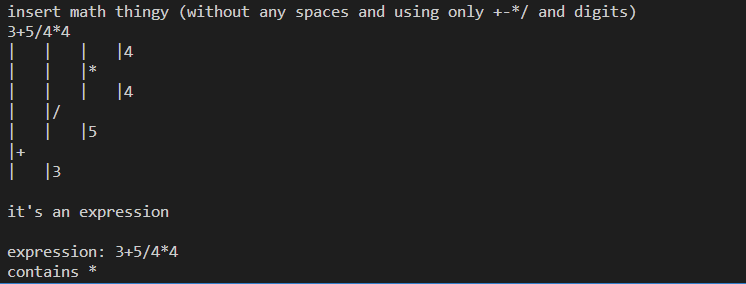


Рисунок 1. Интерфейс программы

# **Тестирование**

Тестирование построения дерева. Как в случае, когда выражение является алгебраическим так и в том случае, когда оно им не является дерево должно быть построено. Тестирование для этих случаев приведено на рисунках 2 и 3.

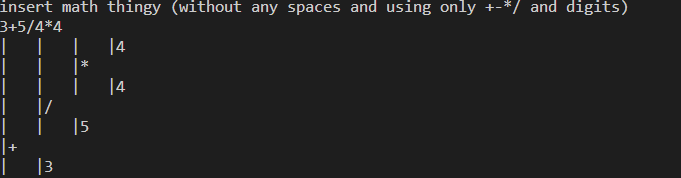


Рисунок 2. Тестирование построения дерева

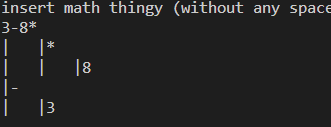


Рисунок 3. Тестирование построения дерева

Тестирование проверки на то, является ли выражение алгебраическим. Проверим для тех же выражений, что и ранее. Результат приведен на рисунках 4 и 5

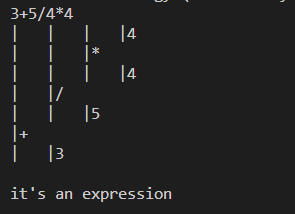


Рисунок 4. Тестирование определения алгебраичности

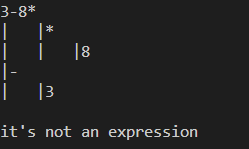


Рисунок 5. Тестирование определения алгебраичности

Тестирование проверки содержания оператора \*. Проверка проводится на случаи, когда такой оператор содержится и не содержится. Также в данную проверку включена проверка восстановления выражения по бинарному дереву. Результаты приведены на рисунках 6 и 7.

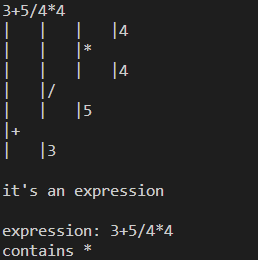


Рисунок 6. Тестирование содержания символа \*

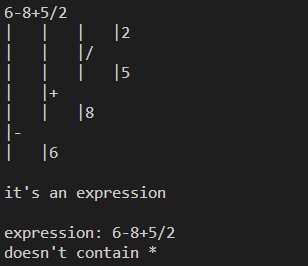


Рисунок 7. Тестирование содержания символа \*

Из результатов выполнения программы видно:

1. Программа работает корректно, все функции работоспособны.

# **Вывод**

В результате выполнения работы я:

1. Освоил базовую работу с бинарными деревьями.
2. Научился программировать создание бинарных деревьев выражений по данному выражению.

# **Исходный код программы**

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  class BinaryTree {  private:      char data;      BinaryTree \*left;      BinaryTree \*right;  public:      BinaryTree(char data) {          this->data = data;          left = nullptr;          right = nullptr;      }      BinaryTree(char data, BinaryTree \*left){          this->data = data;          this->left = left;          right = nullptr;      }      BinaryTree(char data, BinaryTree \*left, BinaryTree \*right) {          this->data = data;          this->left = left;          this->right = right;      }      ~BinaryTree() {          delete left;          delete right;      }      void setLeft(BinaryTree \*left) {          this->left = left;      }      void setRight(BinaryTree \*right) {          this->right = right;      }      BinaryTree \*getLeft() {          return left;      }      BinaryTree \*getRight() {          return right;      }      char getData() {          return data;      }      void addRightNodeRec(const string& s, int it){          if(it >= s.size()){              return;          }          if(it+1 >= s.size()){              right = new BinaryTree(s[it]);              return;          }          right = new BinaryTree( s[it+1], new BinaryTree(s[it]) ); //adding a block of a left operand and an operator          right->addRightNodeRec(s, it+2);          return;      }      bool isExpression(){          if(data == '+' || data == '-' || data == '\*' || data == '/'){ //it's an operator              if(left == nullptr || right == nullptr){                  return false; //if at least one of the operands is missing and the data is an operator then it's not an expression              }              //operator and both operands are present, they must be expressions              return (left->isExpression() && right->isExpression());          }          //not an operator          if(left == nullptr && right == nullptr){              return true; //operand can't have operands so it's an expression          }          return false;      }      bool containsMultiply(){          if(data == '\*'){              return true;          }          if(left != nullptr && left->containsMultiply()){              return true;          }          if(right != nullptr && right->containsMultiply()){              return true;          }          return false;      }      void printTree() {          printTree(0);      }      void printTree(int level) {          if (right != nullptr) {              right->printTree(level + 1);          }          for (int i = 0; i < level; i++) {              cout << "|   ";          }          cout << "|" <<data << endl;          if (left != nullptr) {              left->printTree(level + 1);          }      }      string buildExpression(string s = ""){          if(left != nullptr){              s = left->buildExpression(s);          }          s += data;          if(right != nullptr){              s = right->buildExpression(s);          }          return s;      }  };  int main()  {      setlocale(LC\_ALL, "");      cout << "insert math thingy (without any spaces and using only +-\*/ and digits)" << endl;      string s;      getline(cin, s);      if(s.length() == 0){          cout << "seriosly?" << endl;          return 0;      }      if(s.length() == 1){          BinaryTree \*tree = new BinaryTree(s[0]);          tree->printTree();          delete tree;          return 0;      }      BinaryTree \*tree = new BinaryTree(s[1], new BinaryTree(s[0]));      tree->addRightNodeRec(s, 2);      tree->printTree();      cout << endl;      tree->isExpression() ? cout << "it's an expression" << endl : cout << "it's not an expression" << endl;      cout << endl;      cout << "expression: " << tree->buildExpression() << endl;      tree->containsMultiply() ? cout << "contains \*" << endl : cout << "doesn't contain \*" << endl;      delete tree;      return 0;  } |