Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №7
дисциплины «Алгоритмизация»
Вариант___

Выполнил:
Иващенко Олег Андреевич
2 курс, группа ИВТ-б-о-22-1,
09.03.02 «Информационные и вычислительные машины»,
направленность (профиль)
«Программное обеспечение средств вычислительной техники и автоматизированных систем»

_____
(подпись)

Руководитель практики:
Доцент кафедры инфокоммуникации
Воронкин Роман Александрович

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Тема: «Алгоритм Хаффмана»

Порядок выполнения работы:

Таблица 1 – Код основной программы Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace algorithm7
{
   internal class Program
   {
      static void Main(string[] args)
      {
         string inputString = string.Empty;
         Console.WriteLine("[Program] Введите строку для кодирования");
         Console.Write(">>> "); inputString = Console.ReadLine();

         // Создание дерева Хаффмана и кодирование строки
         HuffmanTree huffmanTree = new HuffmanTree();
         string encodedString = huffmanTree.Encode(inputString);

         // Вывод закодированной строки
         Console.WriteLine($"[Program] Закодированная строка: {encodedString}");

         //Декодирование строки и вывод
         string decodedString = huffmanTree.Decode(encodedString);
         Console.WriteLine($"[Program] Раскодированная строка: {decodedString}");

         Console.ReadKey();
      }
   }
}
```

Таблица 2 – Код класса Huffman.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace algorithm7
{
   class HuffmanNode : IComparable<HuffmanNode>
   {
      public char Character { get; set; }
      public int Frequency { get; set; }
      public HuffmanNode Left { get; set; }
      public HuffmanNode Right { get; set; }

      public bool IsLeaf() { return Left == null && Right == null; }

      public int CompareTo(HuffmanNode other) { return Frequency.CompareTo(other.Frequency); }
   }
```

```
class HuffmanTree
{
    private HuffmanNode root;

    // Кодирование
    public string Encode(string input)
    {
        // Построение дерева и таблицы кодов
        BuildTree(input);
        Dictionary<char, string> codeTable = BuildCodeTable(root);

        StringBuilder encoded = new StringBuilder();
        foreach (char character in input) encoded.Append(codeTable[character]);

        return encoded.ToString();
    }

    // Декодирование строки
    public string Decode(string encodedString)
    {
        StringBuilder decoded = new StringBuilder();
        HuffmanNode currentNode = root;

        foreach (char bit in encodedString)
        {
            if (bit == '0' && currentNode.Left != null) currentNode = currentNode.Left;
            else if (bit == '1' && currentNode.Right != null) currentNode = currentNode.Right;
            else return "[Huffman] Ошибка декодирования";

            if (currentNode.IsLeaf())
            {
                decoded.Append(currentNode.Character);
                currentNode = root;
            }
        }
        return decoded.ToString();
    }

    // Построение дерева Хаффмана
    private void BuildTree(string input)
    {
        // Подсчет частот символов
        Dictionary<char, int> frequencies = new Dictionary<char, int>();
        foreach (char character in input)
        {
            if (frequencies.ContainsKey(character))
            {
                frequencies[character]++;
            }
            else
            {
                frequencies[character] = 1;
            }
        }

        // Построение приоритетной очереди из узлов дерева
        PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<HuffmanNode>();
        foreach (var pair in frequencies)
            priorityQueue.Enqueue(new HuffmanNode { Character = pair.Key, Frequency = pair.Value });

        // Построение дерева Хаффмана
        while (priorityQueue.Count > 1)
```

```
        {
            HuffmanNode left = priorityQueue.Dequeue();
            HuffmanNode right = priorityQueue.Dequeue();
            HuffmanNode parent = new HuffmanNode { Frequency = left.Frequency + right.Frequency, Left = left,
Right = right };
            priorityQueue.Enqueue(parent);
        }
        root = priorityQueue.Count > 0 ? priorityQueue.Dequeue() : null;
    }

    // Построение таблицы кодов
    private Dictionary<char, string> BuildCodeTable(HuffmanNode root)
    {
        Dictionary<char, string> codeTable = new Dictionary<char, string>();
        BuildCodeTableRecursive(root, "", codeTable);
        return codeTable;
    }

    private void BuildCodeTableRecursive(HuffmanNode node, string code, Dictionary<char, string> codeTable)
    {
        if (node != null)
        {
            if (node.IsLeaf()) codeTable[node.Character] = code;

            BuildCodeTableRecursive(node.Left, code + "0", codeTable);
            BuildCodeTableRecursive(node.Right, code + "1", codeTable);
        }
    }
}

class PriorityQueue<T> where T : IComparable<T>
{
    private List<T> heap;

    public int Count => heap.Count;

    public PriorityQueue()
    {
        heap = new List<T>();
    }

    public void Enqueue(T item)
    {
        heap.Add(item);
        int i = heap.Count - 1;
        while (i > 0)
        {
            int parent = (i - 1) / 2;
            if (heap[parent].CompareTo(heap[i]) <= 0)
                break;

            Swap(parent, i);
            i = parent;
        }
    }

    public T Dequeue()
    {
        if (heap.Count == 0) throw new InvalidOperationException("[Huffman] Очередь пуста");

        T root = heap[0];
        heap[0] = heap[heap.Count - 1];
        heap.RemoveAt(heap.Count - 1);
```

```
        int i = 0;
        while (true)
        {
            int leftChild = 2 * i + 1;
            int rightChild = 2 * i + 2;
            int smallestChild = i;

            if (leftChild < heap.Count && heap[leftChild].CompareTo(heap[smallestChild]) < 0)
                smallestChild = leftChild;

            if (rightChild < heap.Count && heap[rightChild].CompareTo(heap[smallestChild]) < 0)
                smallestChild = rightChild;

            if (smallestChild == i)
                break;

            Swap(i, smallestChild);
            i = smallestChild;
        }
        return root;
    }

    private void Swap(int i, int j)
    {
        T temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }
  }
}
```

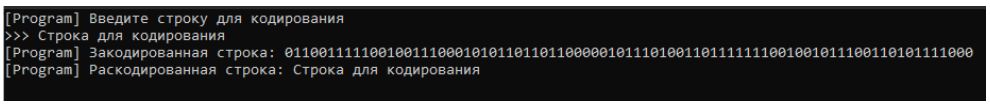

Рисунок 1 – Результат вывода программы

Вывод: В процессе выполнения практической работы была написана программа, кодирующая строку с помощью алгоритма Хаффмана. Суть алгоритма заключается в том, что он подсчитывает частоту встречи каждого символа, строил дерево Хаффмана (по сути граф, в котором каждый лист представляет собой символ, а каждое ребро имеет вес, равный сумме его потомков, при этом символы сортируются по частоте). Далее алгоритм присваивает символам код – либо 0, либо 1, в зависимости от того, как часто встречается символ относительно своей пары. После этого символы заменяются соответствующими им кодам, преобразуя и кодируя строку. По построенной таблице в обратном порядке строка может декодироваться и

вернуться в свой изначальное состояние. Алгоритм используется для сжатия данных, тем самым уменьшая размеры файлов.