**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе № 2**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Алгоритмы кодирования**. Вариант: 2**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Савенко Н.С |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

**Постановка задачи**

Реализовать кодирование и декодирование по алгоритму

Шеннона-Фано входной строки, вводимой через консоль

Посчитать объем памяти, который занимает исходная и закодированная

строки

Выводить на экран таблицу частот и кодов, результат кодирования и

декодирования, коэффициент сжатия

**Описание реализуемых алгоритмов**

Символы первичного алфавита выписывают по

убыванию вероятностей (частот встречаемости)

Символы полученного алфавита делят на две

части, суммарные вероятности символов которых

максимально близки друг другу

В префиксном коде для первой части алфавита

присваивается двоичная цифра «1», второй

части — «0» (или наоборот)

Полученные части рекурсивно делятся и их частям

назначаются соответствующие двоичные цифры в

префиксном коде

Передавать можно кодирующее дерево или

таблицу символов/кодов + закодированную

последовательность

На принимающей стороне требуется побитово

читать строку, составляя путь от корня к узлу

Если передавалась кодирующая таблица, то в

закодированной последовательности должны

присутствовать символы-разделители

**Оценка временной сложности**

1. Encode

O(N)

1. Decode

O(N)

1. EncodeMap

O(N)

1. GetFrequencyDictionary

O(N)

1. SplitByFrequency

O(N)

1. ToString

O(N)

**Описание Unit тестов**

Во всех тестах создается Encoder, затем выполняются нужные операции над ним. Проверяется верность элементов после операций.

**Примеры работы**

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Листинг**

using Laba5.Map;  
  
namespace Laba5;  
  
public class FanoEncoder  
{  
 private IDictionary<char, int>? \_frequencyMap;  
 private IDictionary<char, string>? \_encodedMap;  
 private string Source { get; }  
  
 public IDictionary<char, string> EncodedMap  
 {  
 get  
 {  
 if (\_encodedMap != null)  
 {  
 return \_encodedMap;  
 }  
  
 var dict = new ListMap<char, string>();  
 foreach (var key in FrequencyDictionary.Keys)  
 {  
 dict.Add(key, "");  
 }  
 \_encodedMap = MakeCodeMap(dict, KeysSortedByDesc);  
 return \_encodedMap;  
 }  
 }  
  
 public MemoryStream Encoded  
 {  
 get  
 {  
 var stream = new MemoryStream();  
 using var writer = new BinaryWriter(stream);  
 foreach (var pseudoBit in Source.Select(key => EncodedMap[key]).SelectMany(code => code))  
 {  
 writer.Write(pseudoBit == '1');  
 }  
  
 return stream;  
 }  
 }  
  
 public IDictionary<char, int> FrequencyDictionary  
 {  
 get  
 {  
 if (\_frequencyMap != null)  
 {  
 return \_frequencyMap;  
 }  
  
 \_frequencyMap = new ListMap<char, int>();  
 foreach (var symbol in Source)  
 {  
 if (\_frequencyMap.Keys.Contains(symbol))  
 {  
 \_frequencyMap[symbol]++;  
 }  
 else  
 {  
 \_frequencyMap.Add(symbol, 1);  
 }  
 }  
  
 return \_frequencyMap;  
 }  
 }  
  
 private IEnumerable<char> KeysSortedByDesc => FrequencyDictionary.Keys.OrderByDescending(key => FrequencyDictionary[key]);  
 public double Compress => Encoded.ToArray().Length / (double)(Source.Length \* 16);  
 public FanoEncoder(string source)  
 {  
 Source = source;  
 \_frequencyMap = null;  
 \_encodedMap = null;  
 }  
   
 private IDictionary<char, string> MakeCodeMap(IDictionary<char, string> codeMap, IEnumerable<char> keys)  
 {  
 (IEnumerable<char> leftGroup, IEnumerable<char> rightGroup) = SplitByFrequency(keys);  
 var leftPart = leftGroup as char[] ?? leftGroup.ToArray();  
 var rightPart = rightGroup as char[] ?? rightGroup.ToArray();  
  
 foreach (var c in leftPart)  
 {  
 codeMap[c] = $"1{codeMap[c]}";  
 }  
  
 if (leftPart.Count() > 1)  
 {  
 MakeCodeMap(codeMap, leftPart);  
 }  
  
 foreach (var c in rightPart)  
 {  
 codeMap[c] = $"0{codeMap[c]}";  
 }  
   
 if (rightPart.Count() > 1)  
 {  
 MakeCodeMap(codeMap, rightPart);  
 }  
  
 return codeMap;  
 }  
  
 private (IEnumerable<char>, IEnumerable<char>) SplitByFrequency(IEnumerable<char> keys)  
 {  
 var keysArray = keys as char[] ?? keys.ToArray();  
 var keysList = keysArray.ToList();  
 var groups = (Left: new List<char>(), Right: new List<char>());  
 var sumFrequency = keysArray.Sum(key => FrequencyDictionary[key]);  
 var tmpSum = 0;  
   
 foreach (char key in keysList)  
 {  
 if (tmpSum < sumFrequency / 2)  
 {  
 groups.Left.Add(key);  
 tmpSum += FrequencyDictionary[key];  
 }  
 else  
 {  
 groups.Right.Add(key);  
 }  
 }  
  
 return groups;  
 }  
  
 public string Decode(byte[] bites)  
 {  
 string tempKey = String.*Empty*;  
 string result = String.*Empty*;  
 var reverted = bites.Reverse();  
 foreach (var bite in reverted)  
 {  
 if (EncodedMap.Values.Contains(tempKey))  
 {  
 result += EncodedMap.Keys.First(key => EncodedMap[key] == tempKey);  
 tempKey = $"{bite}";  
 }  
 else  
 {  
 tempKey = $"{bite}{tempKey}";  
 }  
 }  
 if (EncodedMap.Values.Contains(tempKey))  
 {  
 result += EncodedMap.Keys.First(key => EncodedMap[key] == tempKey);  
 }  
  
 return String.*Concat*(result.Reverse());  
 }  
  
 public override string ToString()  
 {  
 return String.*Concat*(Encoded.ToArray());  
 }  
}

using System.Collections;  
  
namespace Laba5.Map;  
  
public class ListMap<TKey, TValue> : IDictionary<TKey, TValue>  
{  
 public ListMap()  
 {  
 Keys = new List<TKey>();  
 Values = new List<TValue>();  
 }  
  
 public int Count { get; }  
 public bool IsReadOnly { get; }  
 public ICollection<TKey> Keys { get; }  
 public ICollection<TValue> Values { get; private set; }  
 public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()  
 {  
 var tmpList = new List<KeyValuePair<TKey, TValue>>();  
 var tmpValues = Values.ToList();  
 var tmpKeys = Keys.ToList();  
 foreach (var key in Keys)  
 {  
 var index = tmpKeys.IndexOf(key);  
 tmpList.Add(new KeyValuePair<TKey, TValue>(key, tmpValues[index]));  
 }  
  
 return tmpList.GetEnumerator();  
 }  
  
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return GetEnumerator();  
 }  
  
 public void Add(KeyValuePair<TKey, TValue> item)  
 {  
 throw new NotImplementedException();  
 }  
  
 public void Clear()  
 {  
 throw new NotImplementedException();  
 }  
  
 public bool Contains(KeyValuePair<TKey, TValue> item)  
 {  
 throw new NotImplementedException();  
 }  
  
 public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex)  
 {  
 throw new NotImplementedException();  
 }  
  
 public bool Remove(KeyValuePair<TKey, TValue> item)  
 {  
 throw new NotImplementedException();  
 }  
  
  
 public void Add(TKey key, TValue value)  
 {  
 Keys.Add(key);  
 Values.Add(value);  
 }  
  
 public bool ContainsKey(TKey key)  
 {  
 throw new NotImplementedException();  
 }  
  
 public bool Remove(TKey key)  
 {  
 throw new NotImplementedException();  
 }  
  
 public bool TryGetValue(TKey key, out TValue value)  
 {  
 throw new NotImplementedException();  
 }  
  
 public TValue this[TKey key]  
 {  
 get  
 {  
 var index = Keys.Select( (item, index) => new {Item = item, Index = index}).First(i => i.Item.Equals(key)).Index;  
 return Values.ToList()[index];  
 }  
 set  
 {  
 var tmpKeys = Keys.ToList();  
 var tmpValues = Values.ToList();  
 var index = tmpKeys.IndexOf(key);  
 tmpValues[index] = value;  
 Values = tmpValues;  
 }  
 }  
}