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 КОДИРОВАНИЕ

ВАРИАНТ: 2

Студент гр. 8309 _____ Иванов Д.К.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Исходная формулировка задания:

Необходимо реализовать кодирование и декодирование по Шеннону-Фано, вместе с оценкой сжатия в %.

Организация данных:

Название	Описание работы метода	Оценка временной сложности
Coding	Используется для кодирования символов на основе посимвольного состава входящей строки. Возвращает string с закодированной входной строкой	$O(kn + t + n^2)$ k-размер входящей строки t-размер предыдущего алфавита
Coding_info	Используется для кодирования символов на основе посимвольного состава входящей строки, а так же оценке сжатия. Не возвращает ничего, но выводит информацию в консоль	$O(kn + t + n^2)$
Input_Code	Используется для вывода алфавита кодировок в консоль.	O(n)
Decoding	Используется для декодирования входящей строки на основе уже имеющегося алфавита. Возвращает string с декодированной входной строкой.	O(k*n)
Decoding_info	Используется для декодирования входящей строки на основе уже имеющегося алфавита. Не возввращает ничего, но выводит информацию в консоль.	O(k*n)

Описание реализованных unit-тестов

Название Unit-теста	Описание работы
TEST_METHOD(TestEncode)	Проверка кодирования элементов
TEST_METHOD(TestDecode)	Проверка декодирования элементов

Программа

Algoritm.h

```
#pragma once

#include <C:\Users\user\Desktop\lab2\lab2\Map.h>

class Alg_Shanon_Fano
{
public:
	string coding(string tocoding);
	string coding_Info(string tocoding);
	void Input_Code();
	string decoding(string todecoding);
	string decoding_info(string todecoding);

private:
	struct Node
	{
		string name;
		string code;
		unsigned int amount;
		Node* _1;
		Node* _0;
	};

	void coder(Node& node, string code);

	List<Node> alphabet;
};

string Alg_Shanon_Fano::coding(string input)
{
	Map<char, unsigned int> composition;
	for (unsigned int i = 0; i < input.size(); ++i)
	{
		char temp = input.at(i);

		if (composition.find(temp) == NULL)
			composition.insert(temp, 1);
		else
			composition.change(temp, composition.find(temp) + 1);
	}
	alphabet.clear();

	List<char> names;
	composition.get_keys(names);
	List<unsigned int> weights;
	composition.get_values(weights);

	for (unsigned int i = 0; i < names.GetSize(); ++i)
	{
		Node temp;
		temp.name = names[i];
		temp.amount = weights[i];
		alphabet.push_back(temp);
	}

	if (alphabet.GetSize() <= 2)
	{
		for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
			alphabet[i].code = i;
```

```
	string answer;
	while (input.size() > 0)
		for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
			if (alphabet[i].name[0] == input[0])
			{
				answer += alphabet[i].code;
				input.erase(0, 1);
				break;
			}
	return answer;
}

List<Node> allNodes;

while (alphabet.GetSize() > 1)
{
	Node* smallest1;
	Node* smallest2;
	if (alphabet[0].amount < alphabet[1].amount)
	{
		smallest1 = &alphabet[0];
		smallest2 = &alphabet[1];
	}
	else
	{
		smallest1 = &alphabet[1];
		smallest2 = &alphabet[0];
	}

	for (unsigned int i = 2; i < alphabet.GetSize(); ++i)
	{
		if (alphabet[i].amount < smallest1->amount)
		{
			smallest2 = smallest1;
			smallest1 = &alphabet[i];
		}
		else if (alphabet[i].amount < smallest2->amount)
			smallest2 = &alphabet[i];
	}
	Node temp;

	temp.name = smallest2->name;
	temp.name.append(smallest1->name);

	allNodes.push_back(*smallest2);
	temp._1 = &allNodes[allNodes.GetSize() - 1];

	allNodes.push_back(*smallest1);
	temp._0 = &allNodes[allNodes.GetSize() - 1];

	temp.amount = temp._0->amount + temp._1->amount;

	if (smallest1 < smallest2)
	{
		for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
			if (&alphabet[i] == smallest2)
				alphabet.removeAt(i);
		for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
			if (&alphabet[i] == smallest1)
				alphabet.removeAt(i);
	}
	else
	{
```

```
                        for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
                                if (&alphabet[i] == smallest1)
                                        alphabet.removeAt(i);
                        for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
                                if (&alphabet[i] == smallest2)
                                        alphabet.removeAt(i);
                }

                alphabet.push_back(temp);
        }

        coder(alphabet[0], "");

        alphabet.clear();



        for (unsigned int i = 0; i < allNodes.GetSize(); i++)
                if (allNodes[i].name.size() == 1)
                        alphabet.push_back(allNodes[i]);

        string answer;
        while (input.size() > 0)
                for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
                        if (alphabet[i].name[0] == input[0])
                        {
                                answer += alphabet[i].code;
                                input.erase(0, 1);
                                break;
                        }
        return answer;
}

string Alg_Shanon_Fano::coding_Info(string input)
{
        Map<char, unsigned int> composition;
        for (unsigned int i = 0; i < input.size(); ++i)
        {
                char temp = input.at(i);

                if (composition.find(temp) == NULL)
                        composition.insert(temp, 1);
                else
                        composition.change(temp, composition.find(temp) + 1);
        }
        cout << input << endl;
        cout << "Contains:" << endl;
        composition.input_List();

        string answer = coding(input);

        short preWeight = input.size() * 8;
        double postweight = answer.size();
        auto rate = (1 - postweight / preWeight) * 100;
        cout << "Initial text weight: " << preWeight << " bytes, code weight: " << postweight << "
bytes, compression rate: " << rate << " %" << endl;

        cout << endl << "codingd:" << endl << answer << endl;
        return answer;
}

void Alg_Shanon_Fano::Input_Code()
{
        for (unsigned int i = 0; i < alphabet.GetSize(); i++)
```

```
            cout << alphabet[i].name << " - " << alphabet[i].code << endl;
}

string Alg_Shanon_Fano::decoding(string input)
{
	if (alphabet.GetSize() == 0)
		throw exception("Attempt to decoding without alphabet");
	string current;
	string answer;
	while (input.size() > 0)
	{
		current += input[0];
		input.erase(0, 1);
		for (unsigned int i = 0; i < alphabet.GetSize(); ++i)
			if (alphabet[i].code == current)
			{
				answer += alphabet[i].name;
				current.clear();
				if (input.size() == 0)
					return answer;
			}

	}
	throw exception("Couldn't decoding");
}

string Alg_Shanon_Fano::decoding_info(string input)
{
	string answer = decoding(input);
	short preWeight = answer.size() * 8;
	double postweight = input.size();
	auto rate = (1 - postweight / preWeight) * 100;
	cout << "Initial text weight: " << preWeight << " bytes, code weight: " << postweight << "
bytes, compression rate: " << rate << " %" << endl;

	cout << endl << "decodingd:" << endl << answer << endl;
	return answer;
}

void Alg_Shanon_Fano::coder(Node& node, string code)
{
	if (node.name.size() == 1)
	{
		node.code = code;
		return;
	}
	string temp0 = code;
	temp0.append("0");
	coder(*node._0, temp0);
	string temp1 = code;
	temp1.append("1");
	coder(*node._1, temp1);
}
```

List.h

```
#pragma once

using namespace std;

#include <iostream>

template<typename T>
class List
```

```
{
public:
	List();
	~List();

	void pop_front();
	void pop_back();
	void push_back(T data);
	void push_front(T data);
	void insert(T value, int index);
	void clear();
	void removeAt(int index);
	unsigned int GetSize()
	{
		return Size;
	}
	T& operator[] (const int index);
	void print_to_console();
	void set(T value, int index);
	bool isEmpty();


	int search_lastest(List<T>& search);		// Compare 2 lists. Get first element (bigger list, not in ()) index where Comparison starts, or get -1 if there is no Comparison.
private:


	template<typename T>
	class Node
	{
	public:
		Node* pNext;				// Next element
		Node* pPrev;			// pPrev element
		T data;

		Node(T data = T(), Node* pNext = nullptr, Node* pPrev = nullptr)		// By default
		{
			this->data = data;
			this->pNext = pNext;
			this->pPrev = pPrev;
		}
	};
	int Size;
	Node<T>* head;
	Node<T>* tail;
};


template<typename T>
List<T>::List()		// Constructor
{
	Size = 0;		// when just created, list size is always 0
	head = nullptr;		// by default, next and prev pointers are nullptr
	tail = nullptr;
}


template<typename T>
List<T>::~List()		// Destructor
{
	clear();
}
```

```
template<typename T>
void List<T>::pop_front()
{
	if (Size == 0)
		throw exception("List is empty, pop_front() didn't work");
	Node<T>* temp = head;
	head = head->pNext;
	delete temp;
	Size--;
}


template<typename T>
void List<T>::pop_back()
{
	if (Size == 0)
		throw exception("List is empty, pop_back() didn't work");
	removeAt(Size - 1);
}

template<typename T>
void List<T>::push_back(T data)
{
	if (head == nullptr)
	{
		head = new Node<T>(data);
		tail = head;
	}
	else
	{
		Node<T>* temp = new Node<T>(data);
		temp->pNext = nullptr;
		temp->pPrev = tail;
		tail->pNext = temp;
		tail = temp;
	}
	Size++;
}


template<typename T>
void List<T>::push_front(T data)
{
	head = new Node<T>(data, head);
	Size++;
	if (Size == 1)
		tail = head;
}


template<typename T>
void List<T>::insert(T data, int index)
{
	if (index == 0)
		push_front(data);
	else
	{
		if (index < 0)
			throw exception("Index (insert(data, index)) is negative");
		if (index > Size)
			throw exception("Index (insert(data, index)) is bigger than list size + 1");
		Node<T>* pPrev = this->head;
		Node<T>* nex = this->head;
```

```
		for (int i = 0; i < index - 1; i++)
		{
			pPrev = pPrev->pNext;
			nex = nex->pNext;
		}
		Node<T>* newNode = new Node<T>(data, pPrev->pNext);
		pPrev->pNext = newNode;

		nex = nex->pNext;
		nex->pPrev = pPrev;
		if (index != Size)
		{
			pPrev = pPrev->pNext;
			nex = nex->pNext;
			nex->pPrev = pPrev;
		}
		else
			tail = nex;

		Size++;
	}
}


template<typename T>
void List<T>::clear()
{
	while (Size)
		pop_front();
}


template<typename T>
void List<T>::removeAt(int index)
{
	if (index == 0)
		pop_front();
	else
	{
		if (index < 0)
			throw exception("Index (removeAt(index)) is negative");
		if (index >= Size)
			throw exception("Index (removeAt(index)) is bigger than list size");
		Node<T>* pPrev = this->head;
		for (int i = 0; i < index - 1; i++)
			pPrev = pPrev->pNext;
		Node<T>* toDelete = pPrev->pNext;
		if (index != Size - 1)
		{
			Node<T>* nex = toDelete->pNext;
			nex->pPrev = pPrev;
		}
		else
			tail = pPrev;
		pPrev->pNext = toDelete->pNext;
		delete toDelete;
		Size--;
	}
}


template<typename T>
T& List<T>::operator[](const int index)
{
```

```
	int counter = 0;
	Node<T>* current = this->head;
	while (current != nullptr)
	{
		if (counter == index)
			return current->data;
		current = current->pNext;
		counter++;
	}
}

template<typename T>
void List<T>::print_to_console()
{
	Node<T>* current = this->head;
	if (Size == 0)
		cout << "List is empty";
	else
		while (current != nullptr)
		{
			cout << current->data << endl;
			current = current->pNext;
		}
}

template<typename T>
void List<T>::set(T data, int index)
{
	if (index < 0)
		throw exception("Index (set(data, index)) is negative");
	if (index >= Size)
		throw exception("Index (set(data, index)) is bigger than list size");
	int counter = 0;
	Node<T>* current = this->head;
	while (current != nullptr)
	{
		if (counter == index)
			break;
		current = current->pNext;
		counter++;
	}

	current->data = data;

}

template<typename T>
bool List<T>::isEmpty()
{
	if (Size == 0)
		return true;
	else
		return false;
}

template<typename T>
int List<T>::search_lastest(List<T>& search)
{
	if (Size == 0)
		throw exception("Main list contains 0 items, findlast() didn't work");
	if (search.GetSize() > Size)
		throw exception("Included list is bigger than main one, findlast() didn't work");
	if (search.GetSize() == 0)
		throw exception("Included list contains 0 items, findlast() didn't work");
```

```
	Node<T>* field = this->tail;
	int Count_step = 0;
	bool Comparison = false;
	for (int i = Size - 1; i >= 0; i--)
	{
		if (field->data == search[search.GetSize() - 1])
		{
			Count_step = 0;
			for (int j = search.GetSize() - 2; j >= 0; j--)
			{
				Count_step++;
				field = field->pPrev;
				if (field->data != search[j])
				{
					Comparison = false;
					for (int k = 0; k < Count_step; k++)
						field = field->pNext;
					Count_step = 0;
					break;
				}
				if (j == 0)
					Comparison = true;
			}
			if ((Comparison == true) || (search.GetSize() == 1))
			{
				return (i - Count_step);
			}
		}
		field = field->pPrev;
	}
	return -1;
}
```

Map.h

```
#pragma once
#include <iostream>
#include <Windows.h>
#include <C:\Users\user\Desktop\lab2\lab2\List.h>



#define RED 0
#define BLACK 1




using namespace std;

template <typename T0, typename T1>
class Map
{
public:
```

```
  Map();
  ~Map();

  unsigned int GetSize()
  {
        return size;
  }

  void insert(T0 key, T1 value);
  void remove(T0 key);
  T1 find(T0 key);
  bool change(T0 key, T1 NewValue);
  void clear();
  void input_Tree();
  void input_List();

  void Paint(int text, int background)
  {
        HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
        SetConsoleTextAttribute(hStdOut, (WORD)((background << 4) | text));
  }

  void get_keys(List<T0>& map)
  {
        map.clear();
        if (size == 0)
              return;
        Key_populate(map, root);
  }
  void get_values(List<T1>& map)
  {
        map.clear();
        if (size == 0)
              return;
        Value_populate(map, root);
  }

private:

  template <typename T0, typename T1>
  class Node
```

```
	{
	public:
		Node* parent;
		Node* left;
		Node* right;
		bool color;
		T0 key;
		T1 value;

		Node(T0 key = T0(), T1 value = T1(), Node* parent = nullptr, Node* left =
nullptr, Node* right = nullptr, bool color = RED)
		{
			this->key = key;
			this->value = value;
			this->parent = parent;
			this->left = left;
			this->right = right;
			this->color = color;
		}
	};

	void input_Tree(int index, int places, Node<T0, T1>* q);
	void input_List(Node<T0, T1>* current);

	void insert(Node<T0, T1>* parent, T0 key, T1 value);
	void Check_uncle(Node<T0, T1>* node);

	void remove(Node<T0, T1>* node);
	void removeFIX(Node<T0, T1>* node, bool leafs);

	void clear(Node<T0, T1>* node);

	void Key_populate(List<T0>& lst, Node<T0, T1>* current);
	void Value_populate(List<T1>& lst, Node<T0, T1>* current);


	Node<T0, T1>* root;
	Node<T0, T1>* leaf;
	unsigned int size;
};
```

```
template <typename T0, typename T1>
Map<T0, T1>::Map()
{
  size = 0;
  root = nullptr;
  leaf = new Node<T0, T1>(NULL, NULL, nullptr, nullptr, nullptr, BLACK);
}

template <typename T0, typename T1>
Map<T0, T1>::~Map()
{
  clear();
}



template <typename T0, typename T1>
void Map<T0, T1>::insert(T0 key, T1 value)
{
  if (size == 0)
  {
        root = new Node<T0, T1>(key, value, nullptr, leaf, leaf, BLACK);
        size++;
        return;
  }
  insert(root, key, value);
}



template <typename T0, typename T1>
void Map<T0, T1>::input_Tree()
{
  int index = 0;
  int places = 0;

  if (index < size)
  {
        input_Tree(index, places + 4, root->right);
        Paint(15, 0);
        cout << root->key;
```

```
        Paint(0, 15);

        cout << endl;
        Paint(15, 0);
        cout << root->value;
        Paint(0, 15);
        cout << endl;
        input_Tree(index, places + 4, root->left);
        index++;
  }
}

template<typename T0, typename T1>
void Map<T0, T1>::input_List()
{
  if (size == 0)
        return;
  input_List(root->left);

  cout << root->key << " - " << root->value << endl;

  input_List(root->right);
}

template <typename T0, typename T1>
void Map<T0, T1>::remove(T0 key)
{
  Node<T0, T1>* node = root;
  while (node != leaf)
  {
        if (node->key == key)
        {
              remove(node);
              node = root;
        }
        else if (node->key > key)
              node = node->left;
        else
              node = node->right;
  }
}
```

```
template <typename T0, typename T1>
T1 Map<T0, T1>::find(T0 key)
{
  if (size == 0)
        return NULL;
  Node<T0, T1>* current = root;
  while (current != leaf)
  {
        if (key == current->key)
              return current->value;
        else if (key < current->key)
              current = current->left;
        else
              current = current->right;
  }
  return NULL;
}

template<typename T0, typename T1>
bool Map<T0, T1>::change(T0 key, T1 NewValue)
{
  if (size == 0)
        return false;
  Node<T0, T1>* current = root;
  while (current != leaf)
  {
        if (key == current->key)
        {
              current->value = NewValue;
              return true;
        }

        else if (key < current->key)
              current = current->left;
        else
              current = current->right;
  }
  return false;
}
```

```
template <typename T0, typename T1>
void Map<T0, T1>::clear()
{
  if (size == 0)
        return;
  clear(root->left);
  clear(root->right);
  delete root;
  size--;
}
```

```
template <typename T0, typename T1>
void Map<T0, T1>::input_Tree(int index, int places, Node<T0, T1>* q)
{
  if (index < size)
  {
        if (q != leaf)
              input_Tree(index, places + 4, q->right);
        for (int i = 0; i < places; i++) {
              Paint(0, 15);
              cout << ' ';
        }
        if (q->color == BLACK)
        {
              Paint(15, 0);
              if (q == leaf)
                    cout << "leaf";
              else
              {
                    Paint(15, 0);
                    cout << q->key;
                    Paint(0, 15);
                    cout << endl;
                    for (int i = 0; i < places; i++) {
                          Paint(0, 15);
                          cout << ' ';
                    }
                    Paint(15, 0);
```

```
                cout << q->value;
                Paint(0, 15);
            }
            Paint(0, 15);
            cout << endl;
        }
        else
        {
            Paint(15, 4);
            cout << q->key;
            Paint(0, 15);
            cout << endl;
            for (int i = 0; i < places; i++) {
                Paint(0, 15);
                cout << ' ';
            }
            Paint(15, 4);
            cout << q->value;
            Paint(0, 15);
            cout << endl;
        }


        if (q != leaf)
            input_Tree(index, places + 4, q->left);
        if (q != leaf)
            index++;
    }
}

template<typename T0, typename T1>
void Map<T0, T1>::input_List(Node<T0, T1>* current)
{
    if (current == leaf)
        return;
    input_List(current->left);

    cout << current->key << " - " << current->value << endl;

    input_List(current->right);
}
```

```
template <typename T0, typename T1>
void Map<T0, T1>::insert(Node<T0, T1>* parent, T0 key, T1 value)
{
  if (key == parent->key)
        throw exception("attempt to add an existant key into map");
  else if (key < parent->key)
  {
        if (parent->left != leaf)
              insert(parent->left, key, value);
        else
        {
              parent->left = new Node<T0, T1>(key, value, parent, leaf, leaf);
              size++;
              if (parent->color == RED)
                    Check_uncle(parent->left);
        }
  }
  else
  {
        if (parent->right != leaf)
              insert(parent->right, key, value);
        else
        {
              parent->right = new Node<T0, T1>(key, value, parent, leaf, leaf);
              size++;
              if (parent->color == RED)
                    Check_uncle(parent->right);
        }
  }
}


template <typename T0, typename T1>
void Map<T0, T1>::Check_uncle(Node<T0, T1>* node)
{
  Node<T0, T1>* parent = node->parent;
  Node<T0, T1>* grandparent = parent->parent;

  if (grandparent->left == parent)
  {
```

```
Node<T0, T1>* uncle = grandparent->right;

if (uncle->color == RED)
{
	parent->color = BLACK;
	uncle->color = BLACK;
	if (grandparent == root)
		return;
	grandparent->color = RED;
	if (grandparent->parent->color == RED)
		Check_uncle(grandparent);
}
else
{


	if (parent->right == node)
	{
		parent->right = node->left;
		parent->right->parent = parent;
		parent->parent = node;
		node->parent = grandparent;
		node->left = parent;
		grandparent->left = node;

		node = parent;
		parent = parent->parent;
	}


	grandparent->left = parent->right;
	if (grandparent->left != leaf)
		grandparent->left->parent = grandparent;
	grandparent->left->parent = grandparent;
	parent->right = grandparent;

	parent->parent = grandparent->parent;

	if (grandparent != root)
	{
		if (grandparent->parent->left == grandparent)
```

```
                        grandparent->parent->left = parent;
                else
                        grandparent->parent->right = parent;
        }
        else
                root = parent;

        grandparent->parent = parent;

        parent->color = BLACK;
        grandparent->color = RED;


    }
}
else
{


    Node<T0, T1>* uncle = grandparent->left;

    if (uncle->color == RED)
    {
        parent->color = BLACK;
        uncle->color = BLACK;
        if (grandparent == root)
                return;
        grandparent->color = RED;
        if (grandparent->parent->color == RED)
                Check_uncle(grandparent);
    }
    else
    {
        if (parent->left == node)
        {
                parent->left = node->right;
                parent->left->parent = parent;
                parent->parent = node;
                node->parent = grandparent;
                node->right = parent;
                grandparent->right = node;
```

```
                node = parent;
                parent = parent->parent;
            }

            Node<T0, T1>* Family_member = parent->left;

            grandparent->right = parent->left;
            if (grandparent->right != leaf)
                grandparent->right->parent = grandparent;
            parent->left = grandparent;

            parent->parent = grandparent->parent;

            if (grandparent != root)
            {
                if (grandparent->parent->right == grandparent)
                {
                    grandparent->parent->right = parent;
                }
                else
                    grandparent->parent->left = parent;
            }
            else
                root = parent;

            grandparent->parent = parent;

            parent->color = BLACK;
            grandparent->color = RED;


        }
    }
}

template <typename T0, typename T1>
void Map<T0, T1>::remove(Node<T0, T1>* node)
{
    if (node->left == leaf && node->right == leaf)
    {
```

```
        if (node == root)
        {
                root = nullptr;
                delete node;
                size--;
        }
        else if (node->color == RED)
        {
                if (node->parent->left == node)
                        node->parent->left = leaf;
                else
                        node->parent->right = leaf;
                delete node;
                size--;
        }
        else
                removeFIX(node, true);
}
else if ((node->left != leaf) && (node->right == leaf))
{
        if (node == root)
        {
                root = node->left;
                node->left->parent = nullptr;
                node->left->color = BLACK;
                delete node;
                size--;
        }
        else if (node->color == RED)
        {
                if (node->parent->left == node)
                {
                        node->parent->left = node->left;
                        node->left->parent = node->parent;
                }
                else
                {
                        node->parent->right = node->left;
                        node->left->parent = node->parent;
                }
                delete node;
```

```
			size--;
		}
		else
			removeFIX(node, true);

	}
	else if ((node->right != leaf) && (node->left == leaf))
	{
		if (node == root)
		{
			root = node->right;
			node->right->parent = nullptr;
			node->right->color = BLACK;
			delete node;
			size--;
		}
		else if (node->color == RED)
		{
			if (node->parent->left == node)
			{
				node->parent->left = node->right;
				node->right->parent = node->parent;
			}
			else
			{
				node->parent->right = node->right;
				node->right->parent = node->parent;
			}
			delete node;
			size--;
		}
		else
			removeFIX(node, true);
	}
	else
	{
		Node<T0, T1>* current = node->right;
		while (current->left != leaf)
			current = current->left;
		node->key = current->key;
		node->value = current->value;
```

```
        remove(current);
  }
}

template <typename T0, typename T1>
void Map<T0, T1>::removeFIX(Node<T0, T1>* node, bool leafs)
{


  Node<T0, T1>* parent = node->parent;
  bool nodeLeft;

  if (leafs == true)
  {
        if (node->left != leaf)
        {
              node->left->parent = parent;
              node->left->color = BLACK;
              if (node == parent->left)
                    parent->left = node->left;
              else
                    parent->right = node->left;
              delete node;
              size--;
              return;
        }
        else if (node->right != leaf)
        {
              node->right->parent = parent;
              node->right->color = BLACK;
              if (node == parent->left)
                    parent->left = node->right;
              else
                    parent->right = node->right;
              delete node;
              size--;
              return;
        }
        else
        {
              if (node == parent->left)
```

```
            {
                  nodeLeft = true;
                  parent->left = leaf;
            }
            else
            {
                  nodeLeft = false;
                  parent->right = leaf;
            }
            delete node;
            size--;
      }
}


Node<T0, T1>* grandparent = parent->parent;
Node<T0, T1>* Family_member;
if (leafs == true)
{
      if (nodeLeft)
            Family_member = parent->right;
      else
            Family_member = parent->left;
}
else
{
      if (node == parent->left)
      {
            nodeLeft = true;
            Family_member = parent->right;
      }

      else
      {
            nodeLeft = false;
            Family_member = parent->left;
      }
}

Node<T0, T1>* SL = Family_member->left;
Node<T0, T1>* SR = Family_member->right;
```

```
if (Family_member->color == RED)
{
	parent->parent = Family_member;
	SL->parent = parent;


	if (root == parent)
	{
		root = Family_member;
		Family_member->parent = nullptr;
	}
	else
	{
		if (grandparent->left == parent)
			grandparent->left = Family_member;
		else
			grandparent->right = Family_member;
		Family_member->parent = grandparent;
	}

	if (nodeLeft == true)
	{
		parent->right = SL;
		Family_member->left = parent;
	}
	else
	{
		parent->left = SL;
		Family_member->right = parent;
	}

	parent->color = RED;
	Family_member->color = BLACK;

	Family_member = SL;
	SL = Family_member->left;
	SR = Family_member->right;
}
```

```
	if (parent->color == BLACK && Family_member->color == BLACK && SL->color == BLACK &&
SR->color == BLACK)
	{
		Family_member->color = RED;
		removeFIX(parent, false);
		return;
	}

	if (Family_member->color == BLACK && SL->color == BLACK && SR->color == BLACK &&
parent->color == RED)
	{
		parent->color = BLACK;
		Family_member->color = RED;
		return;
	}

	if (nodeLeft == true)
	{
		if (SL->color == RED && SR->color == BLACK)
		{
			SL->parent = parent;
			parent->right = SL;
			SL->right->parent = Family_member;
			Family_member->left = SL->right;
			SL->right = Family_member;
			Family_member->parent = SL;
			SL->color = BLACK;
			Family_member->color = RED;
			Family_member = SL;
			SL = SL->left;
			SR = Family_member;
		}

		if (SR->color == RED)
		{
			Family_member->color = parent->color;
			parent->color = BLACK;
			SR->color = BLACK;
			parent->right = SL;
			SL->parent = parent;
			Family_member->left = parent;
```

```
                parent->parent = Family_member;
                if (root == parent)
                {
                        root = Family_member;
                        Family_member->parent = nullptr;
                }
                else
                {
                        Family_member->parent = grandparent;
                        if (grandparent->left == parent)
                                grandparent->left = Family_member;
                        else
                                grandparent->right = Family_member;
                }
        }
}
else
{
        if (SR->color == RED && SL->color == BLACK)
        {
                SR->parent = parent;
                parent->left = SR;
                SR->left->parent = Family_member;
                Family_member->right = SR->left;
                SR->left = Family_member;
                Family_member->parent = SR;
                SR->color = BLACK;
                Family_member->color = RED;
                Family_member = SR;
                SL = Family_member;
                SR = SR->right;
        }

        if (SL->color == RED)
        {
                Family_member->color = parent->color;
                parent->color = BLACK;
                SL->color = BLACK;
                parent->left = SR;
                SR->parent = parent;
                Family_member->right = parent;
```

```
                parent->parent = Family_member;
                if (root == parent)
                {
                        root = Family_member;
                        Family_member->parent = nullptr;
                }
                else
                {
                        Family_member->parent = grandparent;
                        if (grandparent->left == parent)
                                grandparent->left = Family_member;
                        else
                                grandparent->right = Family_member;
                }
        }
  }

  leaf->color = BLACK;
  leaf->parent = nullptr;
  leaf->left = nullptr;
  leaf->right = nullptr;
}

template <typename T0, typename T1>
void Map<T0, T1>::clear(Node<T0, T1>* node)
{
  if (node == leaf)
        return;
  clear(node->left);
  clear(node->right);
  delete node;
  size--;
}

template <typename T0, typename T1>
void Map<T0, T1>::Key_populate(List<T0>& lst, Node<T0, T1>* current)
{
  if (current == leaf)
        return;
  Key_populate(lst, current->left);
  lst.push_back(current->key);
```

```
    Key_populate(lst, current->right);
  }

  template <typename T0, typename T1>
  void Map<T0, T1>::Value_populate(List<T1>& lst, Node<T0, T1>* current)
  {
    if (current == leaf)
          return;
    Value_populate(lst, current->left);
    lst.push_back(current->value);
    Value_populate(lst, current->right);
  }
```

Main.cpp

```
#include <iostream>
#include "C:\Users\user\Desktop\lab2\lab2\Algoritm.h"

using namespace std;

int main()
{
	system("color F0");
	string input = "string for test";
	Alg_Shanon_Fano encryptor;
	string code = encryptor.coding_Info(input);
	encryptor.Input_Code();
	encryptor.decoding_info(code);
}
```

UnitTest1.

```
#include "pch.h"
#include "CppUnitTest.h"
#include "C:\Users\user\Desktop\lab2\lab2\Main.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(TestEncode)
		{
			Alg_Shanon_Fano encryptor;
			string line = "string for test";
			string encodedline = "11000010100110101000011111110110100110011101100 0";
			Assert::AreEqual(encodedline, encryptor.coding(line));
			line = "";
			encodedline = "";
			Assert::AreEqual(encodedline, encryptor.coding(line));
		}
		TEST_METHOD(TestDecode)
		{
			Alg_Shanon_Fano encryptor;
```

```
			string line = "1100001010011010100001111111011010011001110110000";
			string encodedline = "string for test";

			try
			{
				encryptor.decoding(line);
			}
			catch (const std::exception& ex)
			{
				Assert::AreEqual(ex.what(), "Attempt to decode without alphabet");
			}

			encryptor.coding(encodedline);


			line = "1100111111000111011000000111100011011110101010010";

			try
			{
				encryptor.coding(line);
			}
			catch (const std::exception& ex)
			{
				Assert::AreEqual(ex.what(), "Couldn't decode");
			}

			line = encryptor.coding(encodedline);
			Assert::AreEqual(encodedline, encryptor.decoding(line));
		}
	};

}
```