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»
Вариант №2

Студент гр. 8302 _____ Никулин Л.А.

Преподаватель _____ Тутуева А.В.

1.Цель работы

Реализовать кодирование и декодирование по алгоритму Шеннона-Фано.

2.Описание реализуемого класса и методов

Map	основной класс, в котором будут реализовываться функции
List	класс списка, который будет хранить по одному элементу
Shannon	основной класс, в котором будет реализован алгоритм Шеннона-Фано
List<bool> getEncodedString()	метод, который возвращает список нулей и единиц, получившихся в результате кодирования
List<char> decode(std::string)	метод, осуществляющий декодирование нулей и единиц
void setStringForShannon(std::string)	метод, принимающий строку, кодирование и осуществляет ее кодирование
void showInfo()	метод, который будет выводить всю информацию о кодировании строки, занесенную функцией void setStringForShannon(std::string)

3.Оценка временной сложности каждого метода

List<char> decode(std::string)	O(n)
List<bool> getEncodedString()	O(1)
void setStringForShannon(std::string)	O(n)
void showInfo()	O(n)

4.Описание реализованных Unit-тестов

getEncodedString_test	проверяет работу метода getEncodedString, сверяя возвращаемый код кодирования с ожидаемым кодом
decode_test	проверяет работу метода decode, сверяя декодированное слово с ожидаемым
setStringForShannon_test	проверяет работу метода setStringForShannon, путем сверяя входное слово и переменную input_string класса ShannonFano

5.Пример работы программы

```
Входная строка: Классика - то, что каждый считает нужным прочесть и никто не читает.
Количество символов: 68
Закодированная строка: 110010 110011 0100 0110 0110 0101 1010 0100 000 110100 000 001 0111 110101 000 10000 001 0111 000 1010 0100 1011 11011 11000 111000 000 0110 10000 0101 001 0100 10001 001 000 1001 111001 1
011 1001 11000 111010 000 111011 111100 0111 10000 10001 0110 001 111101 000 0101 000 1001 0101 1010 001 0111 000 1001 10001 000 10000 0101 001 0100 10001 001 11111
Количество бит во входной строке: 544
Количество бит в закодированной строке: 285
Коэффициент сжатия:52.3897 %
Символ  Частота встречаемости   Код символа
 " "             11              000
 "т"             8               001
 "а"             5               0100
 "и"             5               0101
 "с"             4               0110
 "о"             4               0111
 "ч"             4               10000
 "е"             4               10001
 "н"             4               1001
 "к"             3               1010
 "ж"             2               1011
 "ы"             2               11000
 "К"             1               110010
 "л"             1               110011
 "-"             1               110100
 ","             1               110101
 "д"             1               11011
 "й"             1               111000
 "у"             1               111001
 "м"             1               111010
 "п"             1               111011
 "р"             1               111100
 "ь"             1               111101
 "."             1               11111
Для продолжения нажмите любую клавишу . . .
```

```
Входная строка: Плох тот классик, которому ничего не приписывают.
Количество символов: 49
Закодированная строка: 110000 10000 0000 110001 0001 001 0000 001 0001 0101 10000 10001 011 011 0100 0101 11001 0001 0101 0000 001 0000 1001 0000 110100 110101 0001 10100 0100 11011 10101 111000 0000 0001 10100
10101 0001 1011 1001 0100 1011 0100 011 111001 11101 10001 11110 001 11111
Количество бит во входной строке: 392
Количество бит в закодированной строке: 214
Коэффициент сжатия:54.5918 %
Символ  Частота встречаемости   Код символа
 "о"             6               0000
 " "             6               0001
 "т"             4               001
 "и"             4               0100
 "к"             3               0101
 "с"             3               011
 "л"             2               10000
 "а"             2               10001
 "р"             2               1001
 "н"             2               10100
 "е"             2               10101
 "п"             2               1011
 "П"             1               110000
 "х"             1               110001
 ","             1               11001
 "м"             1               110100
 "у"             1               110101
 "ч"             1               11011
 "г"             1               111000
 "ы"             1               111001
 "в"             1               11101
 "ю"             1               11110
 "."             1               11111
Для продолжения нажмите любую клавишу . . .
```

```
Входная строка: Классика - это то, что каждый хотел бы прочесть, по возможности - не читая.
Количество символов: 75
Закодированная строка: 11001 10010 0101 0110 0110 0111 10011 0101 000 10100 000 110100 0100 001 000 0100 001 10101 000 10000 0100 001 000 10011 0101 10110 110101 10111 110110 000 110111 001 0100 10001 10010 000
111000 10111 000 110000 111001 001 10000 10001 0110 0100 111010 10101 000 110000 001 000 111011 001 111100 111101 001 10110 110001 001 0110 0100 0111 000 10100 000 110001 10001 000 10000 0111 0100 0101 111110 11
1111
Количество бит во входной строке: 600
Количество бит в закодированной строке: 329
Коэффициент сжатия:54.8333 %
Символ  Частота встречаемости   Код символа
 " "             13              000
 "о"             9               001
 "т"             7               0100
 "а"             4               0101
 "с"             4               0110
 "к"             3               0111
 "и"             3               10000
 "ч"             3               10001
 "е"             3               10010
 "л"             2               10011
 "к"             2               10100
 "."             2               10101
 ","             2               10110
 "ж"             2               10111
 "ы"             2               110000
 "п"             2               110001
 "н"             2               11001
 "К"             1               110100
 "э"             1               110101
 "д"             1               110110
 "й"             1               110111
 "х"             1               111000
 "б"             1               111001
 "р"             1               111010
 "ь"             1               111011
 "в"             1               111100
 "з"             1               111101
 "м"             1               111110
 "я"             1               111111
 "-"             1
Для продолжения нажмите любую клавишу . . .
```

```cpp
#include <iostream>
#include "Shannon.h"


void showInfoAboutString(Shannon& obj, string str)
{
    obj.Convertion_input_string(str);
    obj.show_on_display();
}

int main()
{
    setlocale(0, "Rus");
    Shannon example;
    showInfoAboutString(example, "Классика – то, что каждый считает нужным прочесть и никто не читает.");
    showInfoAboutString(example, "Плох тот классик, которому ничего не приписывают.");
    showInfoAboutString(example, "Классика – это то, что каждый хотел бы прочесть, по возможности – не читая.");
    system("pause");
}
```

Листинг

Laba2.cpp:

```
#include <iostream>
#include "Shannon.h"


void showInfoAboutString(Shannon& obj, string str)
{
	obj.Convertion_input_string(str);
	obj.show_on_display();
}

int main()
{
	setlocale(0, "Rus");
	Shannon example;
   showInfoAboutString(example, "Классика – то, что каждый считает нужным прочесть и никто не читает.");
	showInfoAboutString(example, "Плох тот классик, которому ничего не приписывают.");
	showInfoAboutString(example, "Классика — это то, что каждый хотел бы прочесть, по возможности — не читая.");
	system("pause");
}
```

Shannon.h:

```
#include <iostream>
#include <string>
#include "List.h"
#include "Map.h"

using namespace std;

class Shannon
{
private:
	List<bool> encoded_string; /* Хранит закодированную строку из нулей и единиц */
	map<char, float, List<bool>> map; /* символ, вероятнотность встречи, код символа */
```

```
	List<bool> code_symbols;  /* Промежуточный список, который хранит код символа при его составлении в set_codes */
	List<pair<char, float>> symbols_and_their_count;	/* символы и их количество в введенной строке */

	void set_codes(size_t indexL, size_t indexR);
public:
	string input_string;
	Shannon() = default;
	void show_on_display();
	List<bool> getEncodedString();
	void Convertion_input_string(string str);
	List<char> decode(string str);
};

List<char> Shannon::decode(string str)
{
	if (input_string.empty())
		throw runtime_error("Input string is empty");

	string encoded_string_to_decode = str;

	if (encoded_string_to_decode.empty())
		throw runtime_error("The string is empty");

	List<char> decoded_string;
	List<bool> temp_code;

	for (int i = 0; i < encoded_string_to_decode.length(); i++)
	{
		temp_code.push_back(encoded_string_to_decode.at(i) - 48);
		if (map.code(temp_code))
		{
			decoded_string.push_back(map.find_key(temp_code));
			temp_code.clear();
		}
	}

	if (!temp_code.isEmpty())
	{
		throw runtime_error("The string is wrong");
	}
```

```
	return decoded_string;
}

void Shannon::Convertion_input_string(string str)
{
	if (str.empty())
		throw runtime_error("Input string is empty");
	input_string = str;

	for (int i = 0; i < input_string.length(); i++)
	{
		map.insert(input_string.at(i), float(1), float(1));
	}

	for (int i = 0; i < input_string.length(); i++)
	{
		symbols_and_their_count.insert_with_sorting(input_string.at(i),
map.find_value1(input_string[i]));
	}

	if (symbols_and_their_count.get_size() == 1)
	{
		code_symbols.push_back(0);
		map.set_value2(input_string.at(0), code_symbols);
	}
	else
		set_codes(0, symbols_and_their_count.get_size());

	for (size_t i = 0; i < input_string.length(); i++)
	{
		encoded_string.push_back(map.find_value2(input_string.at(i)));
	}
}

List<bool> Shannon::getEncodedString()
{
	return encoded_string;
}

void Shannon::show_on_display()
{
	if (input_string.empty())
```

```
            throw runtime_error("There is no string");
        cout << "Входная строка: ";
        cout << input_string << endl;

        cout << "Количество символов: ";
        cout << input_string.length() << endl;

        cout << "Закодированная строка: ";
        for (size_t i = 0; i < input_string.length(); i++)
        {
            map.find_value2(input_string.at(i)).print_to_console();
            cout << " ";
        }
        cout << '\n';
        cout << "Количество бит во входной строке: " << input_string.length() * 8 <<
endl;
        cout << "Количество бит в закодированной строке: " <<
encoded_string.get_size() << endl;
        cout << "Коэффициент сжатия:" << (float)encoded_string.get_size() * 100 /
(input_string.length() * 8) << " %" << endl;

        cout << "Символ" << "\t" << "Частота встречаемости" << "\t" << "Код
символа" << endl;
        for (int i = 0; i < symbols_and_their_count.get_size(); i++)
        {
            cout << " \"" << symbols_and_their_count.at(i).first << "\"\t        " <<
symbols_and_their_count.at(i).second << "            ";

map.find_value2(symbols_and_their_count.at(i).first).print_to_console();
            cout << endl;
        }
}

void Shannon::set_codes(size_t indexL, size_t indexR)
{
        if (indexR - indexL == 2)
        {
            /* Осталось два символа, которые не получили свой код */
            code_symbols.push_back(0);
            map.set_value2(symbols_and_their_count.at(indexL).first,
code_symbols);
            code_symbols.pop_back();
```

```
                code_symbols.push_back(1);
                map.set_value2(symbols_and_their_count.at(indexR - 1).first,
code_symbols);
                code_symbols.pop_back();
                return;
        }
        else if (indexR - indexL == 1)
        {
                /* Остался один символ, который не получил свой код*/
                map.set_value2(symbols_and_their_count.at(indexL).first,
code_symbols);
                return;
        }
        else
        {
                float dS = 0, sum = 0;
                for (int i = indexL; i < indexR; i++)
                {
                        dS += symbols_and_their_count.at(i).second;
                }
                dS /= 2;
                size_t newIndex = indexL;
                for (int i = indexL; i < symbols_and_their_count.get_size(); i++)
                {
                        sum += symbols_and_their_count.at(i).second;
                        newIndex++;
                        if (sum >= dS)
                        {
                                break;
                        }
                }
                code_symbols.push_back(0);
                set_codes(indexL, newIndex);
                code_symbols.pop_back();
                code_symbols.push_back(1);
                set_codes(newIndex, indexR);
                code_symbols.pop_back();
        }
}
```

Map.h:

```
#pragma once
#include <iostream>
#include "List.h"
#include <windows.h>

using namespace std;

enum Color { BLACK, RED, };

template <typename TKey, typename TValue1, class TValue2>
class Node
{
private:
	Color color;
	Node<TKey, TValue1, TValue2>* left, * right, * parent;
	TKey key;
	TValue1 value1;
	TValue2 value2;
public:
	Node() : color(RED), left(nullptr), right(nullptr), parent(nullptr) {}
	Node(TKey key, TValue1 value1) : color(RED), left(nullptr), right(nullptr),
parent(nullptr), key(key), value1(value1) {}
	Node(TKey key, TValue1 value1, TValue2 value2) : color(RED), left(nullptr),
right(nullptr), parent(nullptr), key(key), value1(value1), value2(value2) {}
	bool isOnLeft() { return this == parent->getLeft(); }
	Node<TKey, TValue1, TValue2>* sibling()
	{
		if (parent == nullptr)
			return nullptr;

		if (isOnLeft())
			return parent->right;

		return parent->left;
	}

	bool hasRedChild()
	{
		return (left != nullptr && left->getColor() == RED) || (right != nullptr &&
right->getColor() == RED);
	}
```

```
void setColor(Color color)
{
	this->color = color;
}

Color getColor()
{
	if (this == nullptr)
		return BLACK;
	return color;
}

void setLeft(Node* ptr)
{
	left = ptr;
}

Node* getLeft()
{
	return left;
}

void setRight(Node* ptr)
{
	right = ptr;
}

Node* getRight()
{
	return right;
}

void setParent(Node* parent)
{
	this->parent = parent;
}

Node* getParent()
{
	return parent;
}
```

```
	void setKey(TKey key)
	{
		this->key = key;
	}

	TKey getKey()
	{
		return key;
	}

	void setValue1(TValue1 value)
	{
		this->value1 = value;
	}

	TValue1 getValue1()
	{
		return value1;
	}

	void setValue2(TValue2 value)
	{
		value2.push_front(value);
	}

	TValue2 getValue2()
	{
		return value2;
	}
};
#pragma endregion

#pragma region CLASS_MAP
template <typename TKey, typename TValue1, class TValue2>
class map
{
private:
	Node<TKey, TValue1, TValue2>* root;
	void SetColor(int text, int background)
	{
		HANDLE hConsoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
		SetConsoleTextAttribute(hConsoleHandle, (WORD)((background << 4) | text));
	}

	void fixInsertion(Node<TKey, TValue1, TValue2>*& node)
	{
		if (root == node) {
			node->setColor(BLACK);
			return;
		}
		Color c;
		Node<TKey, TValue1, TValue2>* parent = nullptr, * grandparent = nullptr, * uncle = nullptr;
		//Проверка правил красно чёрного дерева (в случае если родитель красного цвета)
		while (node != root && node->getParent()->getColor() == RED)
		{
			parent = node->getParent();
			grandparent = parent->getParent();
			//В случае если родитель "левый" ребёнок
			if (parent == grandparent->getLeft())
			{
				uncle = grandparent->getRight();
				//В случае если дядя тоже "красный" нам достаточно перекрасить деда, папу и дядю
				if (uncle->getColor() == RED)
				{
					parent->setColor(BLACK);
					grandparent->setColor(RED);
					uncle->setColor(BLACK);
					node = grandparent;
				}
				else
				{
					//Если дядя "чёрный", и наш узел "правый" ребёнок, то нужно сделать его "левым"
					if (node == parent->getRight())
					{
						leftRotate(parent);
						node = parent;
						parent = node->getParent();
					}
```

```
                                        //Также необходимо перекрасить папу и деда,
а также вызвать поворот для деда
                                        rightRotate(grandparent);
                                        c = parent->getColor();
                                        parent->setColor(grandparent->getColor());
                                        grandparent->setColor(c);
                                        node = parent;
                                }
                        }
                        else
                        {
                                //Зеркально отражаем код для другого
расположения папы и дяди
                                uncle = grandparent->getLeft();
                                if (uncle->getColor() == RED)
                                {
                                        parent->setColor(BLACK);
                                        grandparent->setColor(RED);
                                        uncle->setColor(BLACK);
                                        node = grandparent;
                                }
                                else
                                {
                                        if (node == parent->getLeft())
                                        {
                                                rightRotate(parent);
                                                node = parent;
                                                parent = node->getParent();
                                        }
                                        leftRotate(grandparent);
                                        c = parent->getColor();
                                        parent->setColor(grandparent->getColor());
                                        grandparent->setColor(c);
                                        node = parent;
                                }
                        }
                }
                root->setColor(BLACK);
        }

        Node<TKey, TValue1, TValue2>* minValueNode(Node<TKey, TValue1,
TValue2>* node) {
```

```
        Node<TKey, TValue1, TValue2>* ptr = node;

        while (ptr->getLeft() != nullptr)
                ptr = ptr->getLeft();

        return ptr;
}

void leftRotate(Node<TKey, TValue1, TValue2>* node)
{
        Node<TKey, TValue1, TValue2>* right_child = node->getRight();
        node->setRight(right_child->getLeft());

        if (node->getRight() != nullptr)
                node->getRight()->setParent(node);

        right_child->setParent(node->getParent());

        if (node->getParent() == nullptr)
                root = right_child;
        else if (node == node->getParent()->getLeft())
                node->getParent()->setLeft(right_child);
        else
                node->getParent()->setRight(right_child);

        right_child->setLeft(node);
        node->setParent(right_child);
}

void rightRotate(Node<TKey, TValue1, TValue2>* node)
{
        Node<TKey, TValue1, TValue2>* left_child = node->getLeft();
        node->setLeft(left_child->getRight());

        if (node->getLeft() != nullptr)
                node->getLeft()->setParent(node);

        left_child->setParent(node->getParent());

        if (node->getParent() == nullptr)
                root = left_child;
```

```
        else if (node == node->getParent()->getLeft())
                node->getParent()->setLeft(left_child);
        else
                node->getParent()->setRight(left_child);

        left_child->setRight(node);
        node->setParent(left_child);
}

void deleteSubTree(Node<TKey, TValue1, TValue2>* node)
{
        while (node != nullptr) {
                deleteSubTree(node->getLeft());
                deleteSubTree(node->getRight());
                delete node;
                node = nullptr;
        }
        root = nullptr;
}

void setKeys(List<TKey>& list, Node<TKey, TValue1, TValue2>* node)
{
        while (node)
        {
                setKeys(list, node->getLeft());
                list.push_back(node->getKey());
                setKeys(list, node->getRight());
                return;
        }
}

void setValues1(List<TValue1>& list, Node<TKey, TValue1, TValue2>* node)
{
        while (node)
        {
                setValues1(list, node->getLeft());
                list.push_back(node->getValue1());
                setValues1(list, node->getRight());
                return;
        }
}
```

```
	void setValues2(List<TValue2>& list, Node<TKey, TValue1, TValue2>* node)
	{
		while (node)
		{
			setValues2(list, node->getLeft());
			list.push_back(node->getValue2());
			setValues2(list, node->getRight());
			return;
		}
	}
	// возвращает указатель на элемент с помощью ключа
	Node<TKey, TValue1, TValue2>* search(TKey key)
	{
		Node<TKey, TValue1, TValue2>* temp = root;
		while (temp != NULL && key != temp->getKey())
		{
			if (key < temp->getKey()) {
				temp = temp->getLeft();
			}
			else {
				temp = temp->getRight();
			}
		}
		return temp;
	}

public:
	map() : root(nullptr) {}
	~map() {
		deleteSubTree(root);
	}

	//Вставка ноды
	void insert(TKey key, TValue1 value1, TValue1 value2)
	{
		Node<TKey, TValue1, TValue2>* temp = new Node<TKey, TValue1,
TValue2>(key, value1);
		if (!root)
		{
			root = temp;
		}
		else
```

```
		{
			Node<TKey, TValue1, TValue2>* temp1 = root;
			Node<TKey, TValue1, TValue2>* temp2 = nullptr;
			while (temp1 != nullptr)
			{
				temp2 = temp1;
				if (temp1->getKey() < temp->getKey())
				{
					temp1 = temp1->getRight();
				}
				else if (temp1->getKey() == temp->getKey()) //Если
такой символ уже есть, то нужно просто увеличить его количество
				{
					delete temp;
					temp1->setValue1(temp1->getValue1() + value2);
					return;
				}
				else
				{
					temp1 = temp1->getLeft();
				}
			}
			temp->setParent(temp2);
			if (temp2->getKey() <= temp->getKey())
			{
				temp2->setRight(temp);
			}
			else {
				temp2->setLeft(temp);
			}
		}
		fixInsertion(temp);
	}

	//Возвращает значение, используя ключ
	TValue1 find_value1(TKey key)
	{
		if (!root)
			throw invalid_argument("Мапа пуста!!!");
		Node<TKey, TValue1, TValue2>* temp = root;
		while (temp && temp->getKey() != key)
		{
```

```
                if (temp->getKey() < key)
                        temp = temp->getRight();
                else
                        temp = temp->getLeft();
        }
        if (!temp)
                throw invalid_argument("Такого значения нет в мапе!!!");
        return temp->getValue1();
}

TValue2 find_value2(TKey key)
{
        if (!root)
                throw invalid_argument("Мапа пуста!!!");
        Node<TKey, TValue1, TValue2>* temp = root;
        while (temp && temp->getKey() != key)
        {
                if (temp->getKey() < key)
                        temp = temp->getRight();
                else
                        temp = temp->getLeft();
        }
        if (!temp)
                throw invalid_argument("Такого значения нет в мапе!!!");
        return temp->getValue2();
}

//Очистка мапы
void clear() {
        if (root == nullptr)
                throw runtime_error("Мапа пуста!!!");
        deleteSubTree(root);
}

//Возвращает копию списка, содержащую ключи мапы
List<TKey> get_keys()
{
        List<TKey> list;
        setKeys(list, root);
        return list;
}
```

```
//Возвращает копию списка, содержащую значение мапы
List<TValue1> get_values1()
{
	List<TValue1> list;
	setValues1(list, root);
	return list;
}

List<TValue2> get_values2()
{
	List<TValue2> list;
	setValues2(list, root);
	return list;
}

void value1_increase(TKey key, TValue1 value)
{
	Node<TKey, TValue1, TValue2>* temp = search(key);
	temp->setValue1(temp->getValue1() + value);
}

void set_value2(TKey key, TValue2& value)
{
	Node<TKey, TValue1, TValue2>* temp = search(key);
	temp->setValue2(value);
}

TKey find_key(TValue2 code_symbols)
{
	TKey key = NULL;
	bool is_found = false;
	additional_find_key(root, code_symbols, key, is_found);
	return key;
}

bool code(TValue2 code_symbols)
{
	bool is_found = false;
	return additional_code(root, code_symbols, is_found);
}
//Вспомогательная (рекурсивная) функция нахождения ключа
```

```
	void additional_find_key(Node<TKey, TValue1, TValue2>* node, TValue2
code_symbols, TKey& key, bool& is_found)
	{
		if (node)
		{
			if (node->getValue2().compare(code_symbols))
			{
				key = node->getKey();
				is_found = true;
				return;
			}
			else
			{
				additional_find_key(node->getLeft(), code_symbols, key,
is_found);
				if (is_found)
					return;
				additional_find_key(node->getRight(), code_symbols, key,
is_found);
			}
		}
	}

	bool additional_code(Node<TKey, TValue1, TValue2>* node, TValue2
code_symbols, bool& is_found)
	{
		if (node) {
			if (node->getValue2().compare(code_symbols))
			{
				is_found = true;
				return true;
			}
			else
			{
				additional_code(node->getLeft(), code_symbols,
is_found);
				if (is_found)
					return true;
				additional_code(node->getRight(), code_symbols,
is_found);
				return is_found;
			}
```

```
            }
        }

    };
```

List.h:

```
#pragma once
#include <stdexcept>
#include <iostream>
using namespace std;

template<typename T>
class List
{
private:
	class Node
	{
	public:
		T data;
		Node* next, * prev;
	public:
		Node() : next(NULL), prev(NULL) {};
		Node(T data)
		{
			this->data = data;
			next = NULL;
			prev = NULL;
		}

		~Node()
		{
			next = NULL;
			prev = NULL;
		}

		void set_data(T data)
		{
			this->data = data;
		}

		T get_data()
		{
			return data;
		}
```

```
        Node* get_next()
        {
                return next;
        }

        Node* get_prev()
        {
                return prev;
        }

        void set_next(Node* Elem)
        {
                next = Elem;
        }

        void set_prev(Node* Elem)
        {
                prev = Elem;
        }
};
Node* head, * tail;

Node* get_Elem(size_t index)
{
        if (isEmpty() || (index > get_size() - 1))
        {
                throw out_of_range("Invalid argument");
        }
        else if (index == get_size() - 1)
                return tail;
        else if (index == 0)
                return head;
        else
        {
                Node* temp = head;
                while ((temp) && (index--))
                {
                        temp = temp->get_next();
                }
                return temp;
        }
}
```

```
public:
	List() : head(NULL), tail(NULL) {}
	List(int size, int value)
	{
		while (size--)
		{
			push_back(value);
		}
	}

	List(const List<T>& list)
	{
		head = NULL;
		tail = NULL;
		Node* temp = list.head;
		while (temp)
		{
			push_back(temp->get_data());
			temp = temp->get_next();
		}
	}

	~List()
	{
		while (head)
		{
			tail = head->get_next();
			delete head;
			head = tail;
		}
		head = NULL;
	}

	void push_back(T data)
	{
		Node* temp = new Node;
		temp->set_data(data);
		if (head)
		{
			temp->set_prev(tail);
			tail->set_next(temp);
			tail = temp;
```

```
        }
        else
        {
                head = temp;
                tail = head;
        }
}

void push_front(T data)
{
        Node* temp = new Node;
        temp->set_data(data);
        if (head)
        {
                temp->set_next(head);
                head->set_prev(temp);
                head = temp;
        }
        else
        {
                head = temp;
                tail = head;
        }
}

void push_back(List<bool> list)
{
        Node* temp = list.head;
        size_t length = list.get_size();
        while ((temp) && (length--))
        {
                push_back(temp->get_data());
                temp = temp->get_next();
        }
}

void push_front(List& list)
{
        Node* temp = list.tail;
        size_t length = list.get_size();
        while ((temp) && (length--))
        {
```

```
                push_front(temp->get_data());
                temp = temp->get_prev();
        }
}

void pop_back()
{
        if (head != tail)
        {
                Node* temp = tail;
                tail = tail->get_prev();
                tail->set_next(NULL);
                delete temp;
        }
        else if (!isEmpty())
        {
                Node* temp = tail;
                tail = head = NULL;
                delete temp;
        }
        else
                throw out_of_range("Список пуст!!!");
}

void pop_front()
{
        if (head != tail)
        {
                Node* temp = head;
                head = head->get_next();
                head->set_prev(NULL);
                delete temp;
        }
        else if (!isEmpty())
        {
                Node* temp = head;
                head = tail = NULL;
                delete temp;
        }
        else
                throw out_of_range("Список пуст!!!");
}
```

```
void insert(size_t index, T data)
{
	Node* temp;
	temp = get_Elem(index);
	if (temp == head)
		push_front(data);
	else
	{
		Node* newElem = new Node;
		newElem->set_data(data);
		temp->get_prev()->set_next(newElem);
		newElem->set_prev(temp->get_prev());
		newElem->set_next(temp);
		temp->set_prev(newElem);
	}
}

T at(size_t index)
{
	Node* temp;
	temp = get_Elem(index);
	return temp->get_data();
}

void remove(size_t index)
{
	Node* temp;
	temp = get_Elem(index);
	if (temp == head)
		pop_front();
	else if (temp == tail)
		pop_back();
	else
	{
		temp->get_prev()->set_next(temp->get_next());
		temp->get_next()->set_prev(temp->get_prev());
		delete temp;
	}
}

void remove(T data)
```

```
{
	Node* temp = head;
	while (temp && temp->get_data() != data)
		temp = temp->get_next();
	if (!temp)
		throw out_of_range("Недопустимый аргумент!!!");
	if (temp == head)
		pop_front();
	else if (temp == tail)
		pop_back();
	else
	{
		temp->get_prev()->set_next(temp->get_next());
		temp->get_next()->set_prev(temp->get_prev());
		delete temp;
	}
}

size_t get_size()
{
	Node* temp = head;
	size_t length = 0;
	while (temp)
	{
		length++;
		temp = temp->get_next();
	}
	return length;
}

void print_to_console()
{
	Node* temp = head;
	while (temp)
	{
		cout << temp->get_data();
		temp = temp->get_next();
	}
}

void clear()
{
```

```
        while (head)
        {
                tail = head->get_next();
                delete head;
                head = tail;
        }
}

void set(size_t index, T data)
{
        Node* temp;
        temp = get_Elem(index);
        temp->set_data(data);
}

bool isEmpty()
{
        if (!head)
                return true;
        else
                return false;
}

bool compare(List<T> list)
{
        if (get_size() != list.get_size())
                return false;
        for (int i = 0; i < get_size(); i++) {
                if (at(i) != list.at(i))
                        return false;
        }
        return true;
}

bool contains(List<char>& list)
{
        Node* temp = head;
        while (temp) {
                if (temp->get_data().compare(list))
                        return true;
                temp = temp->get_next();
        }
```

```
		return false;
	}

	void nullify()
	{
		head = NULL;
		tail = NULL;
	}

	int get_price(size_t vertex1, size_t vertex2)
	{
		Node* temp = head;
		while (temp)
		{
			if (temp->get_data().vertex1 == vertex1 &&
temp->get_data().vertex2 == vertex2)
				return temp->get_data().cost;
			temp = temp->get_next();
		}
		throw invalid_argument("Некорректный аргумент!!!");
	}

	void insert_with_sorting(char data1, float data2)
	{
		if (!head || head->data.second < data2)
		{
			push_front(pair<char, float>(data1, data2));
			return;
		}
		Node* newElem = new Node;
		newElem->data = pair<char, float>(data1, data2);
		int i = 0;
		Node* temp = head;
		while (temp)
		{
			if (temp->data.second < data2)
			{
				temp->get_prev()->set_next(newElem);
				newElem->set_prev(temp->get_prev());
				temp->set_prev(newElem);
				newElem->set_next(temp);
				return;
```

```
                }
                if (temp->data.first == data1 && temp->data.second == data2) {
                        delete newElem;
                        return;
                }
                i++;
                temp = temp->get_next();
        }
        delete newElem;
        push_back(pair<char, float>(data1, data2));
    }
};
```

TestProject.cpp:

```
#pragma once
#include "CppUnitTest.h"
#include <string>
#include <stdexcept>
#include "../Laba2/Shannon.h"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(Convertion_input_string_test)
		{
			Shannon example;
			string exampleStr = "Блюдо";
			example.Convertion_input_string(exampleStr);
			Assert::AreEqual(example.input_string, exampleStr);

		}
		TEST_METHOD(getEncodedString_test)
		{
			Shannon example;
			string exampleStr = "Блюдо", outString;
			example.Convertion_input_string(exampleStr);
			List<bool> list = example.getEncodedString();
			string TestString = "000001011011"; // "Блюдо"
			for (size_t i = 0; i < list.get_size(); i++)
			{
				if (list.at(i))
					outString = outString + "1";
				else
					outString = outString + "0";
			}
			Assert::AreEqual(TestString, outString);
		}

		TEST_METHOD(decode_test)
```

```
{
	Shannon example;
	string exampleStr = "Чудеса", outString;
	example.Convertion_input_string(exampleStr);
	List<char> list = example.decode("1011101");    //  "сад"
	string TestString = "сад";
	for (size_t i = 0; i < list.get_size(); ++i)
		outString = outString + list.at(i);
	Assert::AreEqual(TestString, outString);
}

TEST_METHOD(convertion_with_empty_string)
{
	Shannon example;
	string exampleStr = "";
	try
	{
		example.Convertion_input_string(exampleStr);

	}
	catch (const exception &ex)
	{
		Assert::AreEqual("Input string is empty", ex.what());
	}
}

TEST_METHOD(decode_with_empty_string)
{
	Shannon example;
	try
	{
		example.show_on_display();

	}
	catch (const exception& ex)
	{
		Assert::AreEqual("There is no string", ex.what());
	}
}

TEST_METHOD(empty_string_in_decode)
{
```

```
                string exampleStr = "";
                Shannon example;
                try
                {
                        example.decode(exampleStr);

                }
                catch (const runtime_error error)
                {
                        Assert::AreEqual("Input string is empty", error.what());
                }
        }
    };
}
```