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: АССОЦИАТИВНЫЙ МАССИВ

ВАРИАНТ: 1

Студент гр. 8309 _____ Иванов Д.К.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Исходная формулировка задания:

Необходимо написать класс и методы для работы с ассоциативными массивами.
Список методов, которые необходимо реализовать:

1. insert(ключ, значение) // добавление элемента с ключом и значением
2. remove(ключ) // удаление элемента дерева по ключу
3. find(ключ) // поиск элемента по ключу
4. clear // очищение ассоциативного массива
5. get_keys // возвращает список ключей
6. get_values // возвращает список значений
7. print // вывод в консоль

Организация данных:

Название	Описание работы метода	Оценка временной сложности
Insert	Добавление элемента в ассоциативный массив. Входящие аргументы представляют собой ключ и связанное значение. В массиве не может быть два элемента с одним и тем же ключом	$O(\log_2 n)$
Remove	Удаление элементов из массива по ключу.	$O(\log_2 n)$
Find	Поиск элемента по ключу. В случае успеха возвращает соответствующее значение, в случае неудачи возвращается NULL.	O(n)
Get_keys	Принимает список в качестве входящего аргумента, очищает его и заполняет ключами в LNR порядке.	O(n)
Get_values	Принимает список в качестве входящего аргумента, очищает его и заполняет значениями ключей в LNR порядке	O(n)
PrintTree	Вывод ключей и их значений в виде красно-чёрного дерева. (RBT)	O(n)
PrintList	Вывод ключей и их значений в виде списка (ассоциативного массива).	O(n)
Clear	Очищение всего массива от элементов	$O(\log_2 n)$
SetColor	Принимает номер цвета текста и цвета фона	O(l)

Описание реализованных unit-тестов

Название Unit-теста	Описание работы
TEST_METHOD(Test InsertFind)	Проверка добавления и посика элемента
TEST_METHOD(Test Remove)	Проверка удаления элемента по ключу
TEST_METHOD(Test Clear)	Проверка очищения массива
TEST_METHOD(Test Get_values_Get_Keys)	Проверка функций Get_values и Get_Keys

Программа

Algoritm.h

```
#pragma once

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

class Algoritm
{
public:
	string encode(string toEncode);
	string encodeReview(string toEncode);
	void printCodes();
	string decode(string toDecode);
	string decodeReview(string toDecode);

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

	void coder(Node& node, string code);

	List<Node> alphabet;
};

string Algoritm::encode(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 Algoritm::encodeReview(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.printList();

		string answer = encode(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 << "Encoded:" << endl << answer << endl;
		return answer;
}

void Algoritm::printCodes()
{
		for (unsigned int i = 0; i < alphabet.GetSize(); i++)
			cout << alphabet[i].name << " - " << alphabet[i].code << endl;
}

string Algoritm::decode(string input)
{
		if (alphabet.GetSize() == 0)
			throw exception("Attempt to decode 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 decode");
}

string Algoritm::decodeReview(string input)
{
		string answer = decode(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 << "Decoded:" << endl << answer << endl;
		return answer;
}
```

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


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

		Node(T data = T(), Node* pNext = nullptr, Node* pPrevious = nullptr)		//
By default
		{
```

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

		nex = nex->pNext;
		nex->pPrevious = previous;
		if (index != Size)
		{
			previous = previous->pNext;
			nex = nex->pNext;
			nex->pPrevious = previous;
		}
		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>* previous = this->head;
		for (int i = 0; i < index - 1; i++)
			previous = previous->pNext;
		Node<T>* toDelete = previous->pNext;
		if (index != Size - 1)
		{
			Node<T>* nex = toDelete->pNext;
			nex->pPrevious = previous;
		}
		else
			tail = previous;
		previous->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>::find_last(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 steps = 0;
		bool match = false;
		for (int i = Size - 1; i >= 0; i--)
		{
			if (field->data == search[search.GetSize() - 1])
			{
				steps = 0;
				for (int j = search.GetSize() - 2; j >= 0; j--)
				{
					steps++;
					field = field->pPrevious;
					if (field->data != search[j])
					{
						match = false;
						for (int k = 0; k < steps; k++)
							field = field->pNext;
						steps = 0;
						break;
					}
					if (j == 0)
						match = true;
				}
				if ((match == true) || (search.GetSize() == 1))
				{
					return (i - steps);
				}
			}
			field = field->pPrevious;
		}
		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 printTree();
	void printList();

	void SetColor(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;
		keyFill(map, root);
	}
	void get_values(List<T1>& map)
	{
		map.clear();
		if (size == 0)
			return;
		valueFill(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 printTree(int index, int spaces, Node<T0, T1>* q);
	void printList(Node<T0, T1>* current);

	void insert(Node<T0, T1>* parent, T0 key, T1 value);
	void uncleCheck(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 keyFill(List<T0>& lst, Node<T0, T1>* current);
	void valueFill(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>::printTree()
{
	int index = 0;
	int spaces = 0;

	if (index < size)
	{
		printTree(index, spaces + 4, root->right);
		SetColor(15, 0);
		cout << root->key;
		SetColor(0, 15);

		cout << endl;
		SetColor(15, 0);
		cout << root->value;
		SetColor(0, 15);
		cout << endl;
		printTree(index, spaces + 4, root->left);
		index++;
	}
}

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

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

	printList(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>::printTree(int index, int spaces, Node<T0, T1>* q)
{
        if (index < size)
        {
            if (q != leaf)
                printTree(index, spaces + 4, q->right);
            for (int i = 0; i < spaces; i++) {
                SetColor(0, 15);
                cout << ' ';
            }
            if (q->color == BLACK)
            {
                SetColor(15, 0);
                if (q == leaf)
                    cout << "leaf";
                else
                {
                    SetColor(15, 0);
                    cout << q->key;
                    SetColor(0, 15);
```

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


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

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

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

    printList(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)
                            uncleCheck(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)
				uncleCheck(parent->right);
		}
	}
}


template <typename T0, typename T1>
void Map<T0, T1>::uncleCheck(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)
				uncleCheck(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)
                        uncleCheck(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>* sibling = 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>* sibling;
if (leafs == true)
{
	if (nodeLeft)
		sibling = parent->right;
	else
		sibling = parent->left;
}
else
{
	if (node == parent->left)
	{
		nodeLeft = true;
		sibling = parent->right;
	}

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

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


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


	if (root == parent)
```

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

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

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

		sibling = SL;
		SL = sibling->left;
		SR = sibling->right;
	}

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

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

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

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

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

```
			}
		}

		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>::keyFill(List<T0>& lst, Node<T0, T1>* current)
{
		if (current == leaf)
			return;
		keyFill(lst, current->left);
		lst.push_back(current->key);
		keyFill(lst, current->right);
}

template <typename T0, typename T1>
void Map<T0, T1>::valueFill(List<T1>& lst, Node<T0, T1>* current)
{
		if (current == leaf)
			return;
		valueFill(lst, current->left);
		lst.push_back(current->value);
		valueFill(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";
		Algoritm encryptor;
		string code = encryptor.encodeReview(input);
		encryptor.printCodes();
		encryptor.decodeReview(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)
		{
			Algoritm encryptor;
			string line = "string for test";
			string encodedline = "1100001010011010100001111111011010011001110110000";
			Assert::AreEqual(encodedline, encryptor.encode(line));
			line = "";
			encodedline = "";
			Assert::AreEqual(encodedline, encryptor.encode(line));
		}
		TEST_METHOD(TestDecode)
		{
			Algoritm encryptor;

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

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

			encryptor.encode(encodedline);


			line = "1100111111000111011000000111100011011110101010010";

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

			line = encryptor.encode(encodedline);
			Assert::AreEqual(encodedline, encryptor.decode(line));
		}
	};

}
```

Вывод

При написании программы были улучшены знания ООП, а также изучена работа алгоритма Дейкстры.