МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ

по курсовой работе
по дисциплине «Алгоритмы и Структуры Данных»
Вариант 3

Студент гр. 8301 _____ Бобров А.И.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

1. Постановка задачи

Дан текстовый файл со строками в формате V1, V2, P, где V1, V2 направленная дуга транспортной сети, а P это её пропускная способность. Исток обозначается как S, а сток как T. Найти максимальный поток в сети реализовав это с помощью алгоритма проталкивания предпотока.

2. Описание реализуемых классов и методов

Flow – класс в котором содержатся основные функции реализованные для поиска максимального потока.

int MaxFlow() – метод для поиска максимального потока в сети

void push(int u, int v) – функция реализующая операцию проталкивания

void lift(int u) – метод используемый для подъёма вершины

void discharge(int u) - применяется к переполненной вершине, для того чтобы протолкнуть поток через допустимые ребра в смежные вершины.

3. Оценка временной сложности

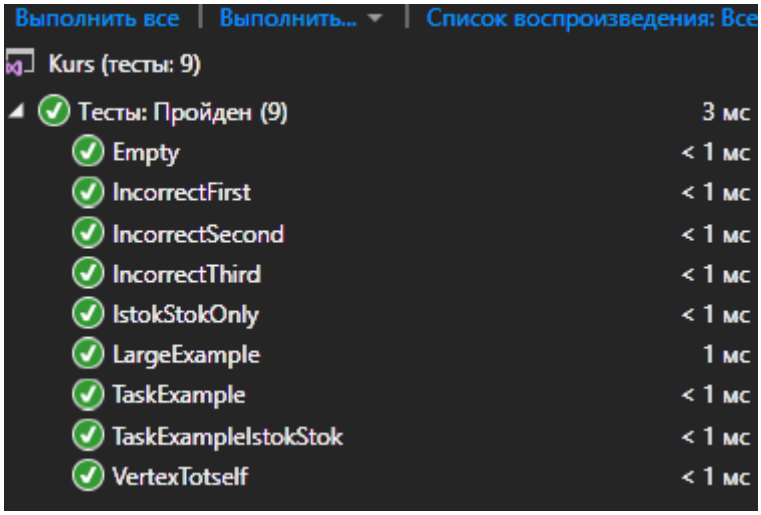
int MaxFlow() - **O(V^2E)**

void push(int u, int v) - **O(1)**

void lift(int u) - **O(V)**

void discharge(int u) - **O(VE)**

4. Описание реализованных Unit-тестов



IncorrectFirst – неправельный ввод V1

IncorrectSecond – неправельный ввод V2

IncorrectThird - неправельный ввод P

Empty – не введены данные

IstokStokOnly – пример только с Истоком и Стоком

LargeExample – пример с большим количеством вершин

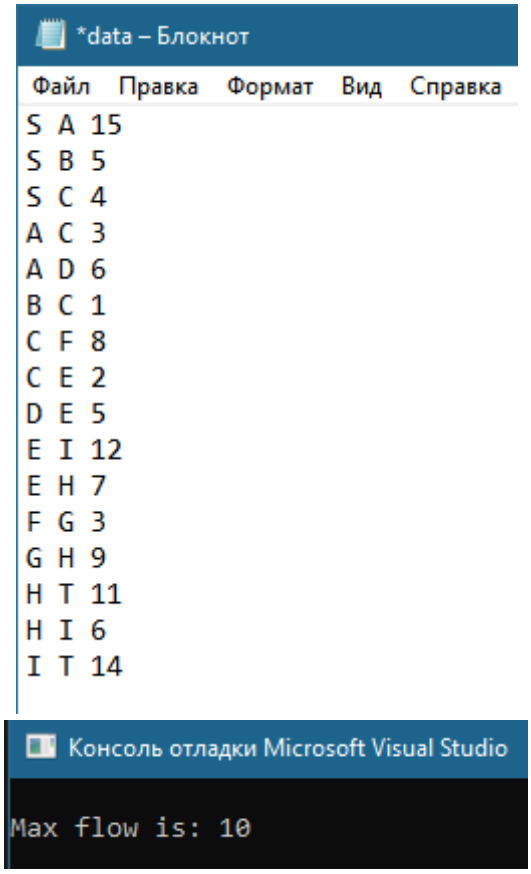
TaskExample – пример из задания

TaskExampleIstokStok – модернизированный пример из задания

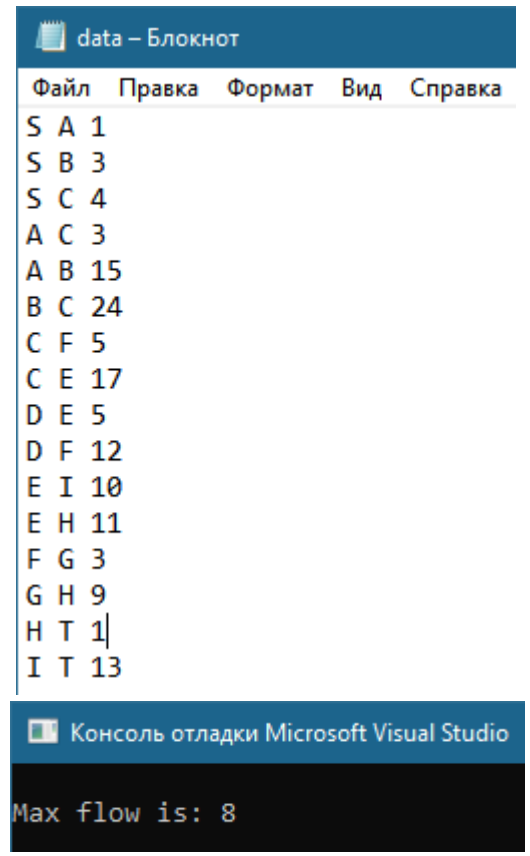
VertexToItself – проверка ребра Исток Исток

5. Примеры работы программы

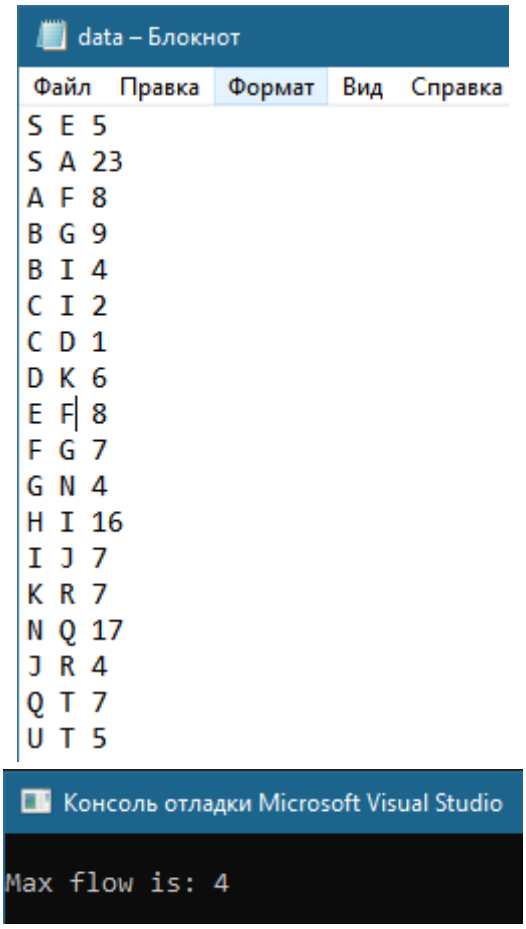
Пример 1:



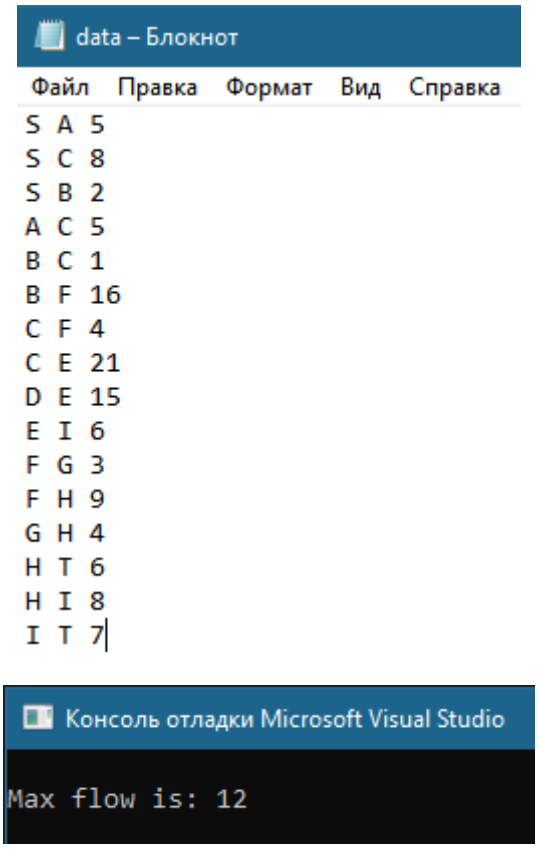
Пример 2:



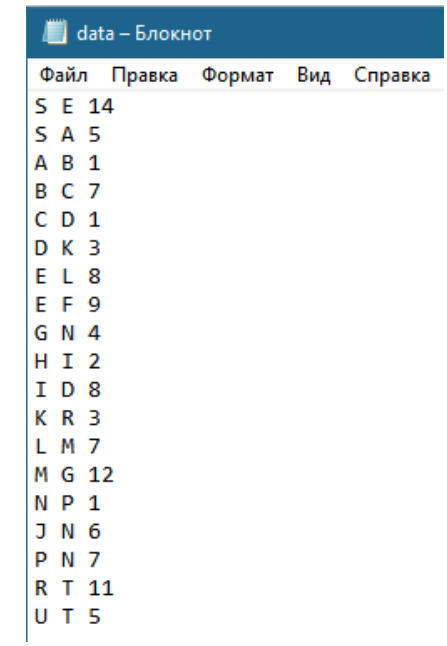
Пример 3:



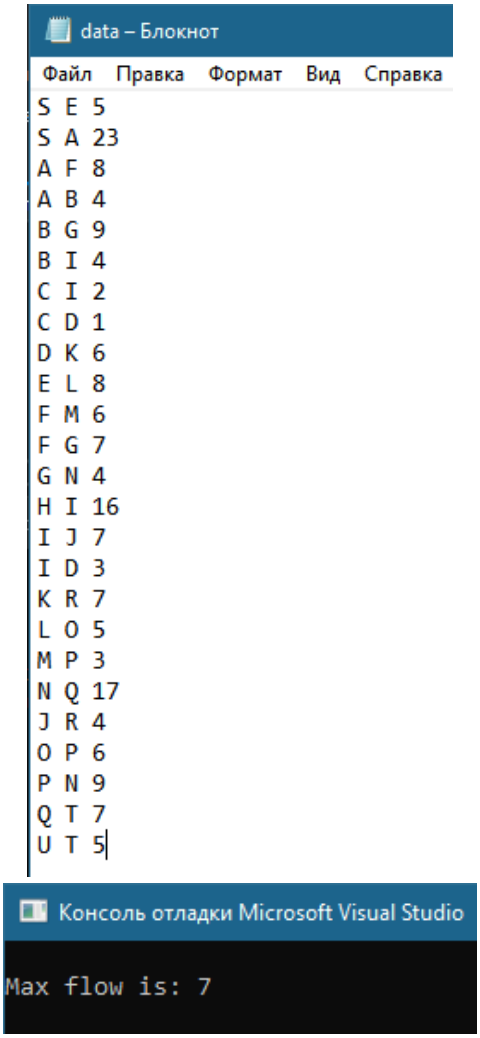
Пример 4:



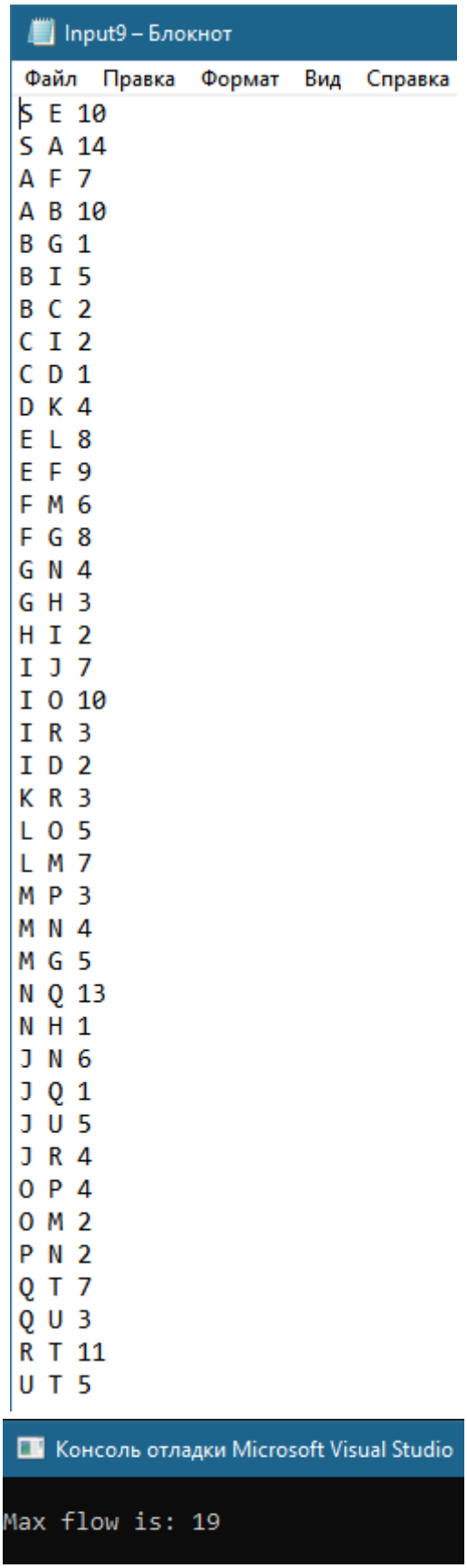
Пример 5:



Пример 6:



Пример 7:


Input9 – Блокнот
Файл Правка Формат Вид Справка
S E 10
S A 14
A F 7
A B 10
B G 1
B I 5
B C 2
C I 2
C D 1
D K 4
E L 8
E F 9
F M 6
F G 8
G N 4
G H 3
H I 2
I J 7
I O 10
I R 3
I D 2
K R 3
L O 5
L M 7
M P 3
M N 4
M G 5
N Q 13
N H 1
J N 6
J Q 1
J U 5
J R 4
O P 4
O M 2
P N 2
Q T 7
Q U 3
R T 11
U T 5
Консоль отладки Microsoft Visual Studio
Max flow is: 19

6. Обоснование выбора используемых структур данных

List используется при переборе вершин сети в методе MaxFlow. List как структура превосходит стандартный массив. За счёт своего функционала он более удобный и с его помощью можно оптимизировать по времени работы функции, в которых он используется.

Map используется для индексирования вершин. Как и в предыдущем случае, позволяет улучшить работу программы.

7. Листинг

Main.cpp:

```
#include <iostream>
#include <fstream>
#include "MaxFlow.h"
int main()
{
	try
	{
		ifstream input("data.txt");
		Flow flow(input);
		std::cout << "\nMax flow is: " << flow.MaxFlow() << "\n";
	}
	catch (exception& exception)
	{
		std::cout << exception.what();
	}
	return 0;
}
```

MaxFlow.h:

```
#pragma once
#include <fstream>
#include "List.h"
#include<string>
#include"Map.h"
using namespace std;

template<typename T>
T min(T a, T b)
{
	return a > b ? b : a;
}

class Flow
{
public:
	//the push-relabel algorithm for the maximal network flow//
	int MaxFlow()
	{
		if (Amount > 2)
		{
			for (int i = 0; i < Amount; i++)
			{
				if (i == s)
					continue;
				e[i] = c[s][i];	c[i][s] += c[s][i];
			}
```

```
		h[s] = Amount;

		List<int> l;
		int current_;
		int current = 0;
		int old;

		for (int i = 0; i < Amount; i++)
			if (i != s && i != t)
				l.push_front(i);
		current_ = l.at(0);

		while (current != l.get_size())
		{
			old = h[current_];
			discharge(current_);
			if (h[current_] != old)
			{
				l.push_front(current_); l.remove(++current);
				current_ = l.at(0); current = 0;
			}
			current++;
			if (current < l.get_size())
				current_ = l.at(current);
		}

		return e[t];
	}

	else
		return c[0][1];
}

//push//
void push(int u, int v)
{
	int f = min(e[u], c[u][v]);
	e[u] -= f;	e[v] += f;
	c[u][v] -= f;	c[v][u] += f;
}

//lift//
void lift(int u)
{
	int min = 2 * Amount + 1;

	for (int i = 0; i < Amount; i++)
		if (c[u][i] && (h[i] < min))
			min = h[i];
	h[u] = min + 1;
}

//discharge//
void discharge(int u)
{
	int V = 0;
	while (e[u] > 0)
	{
		if (c[u][V] && h[u] == h[V] + 1)
		{
			push(u, V); V = 0; continue;
		}
		V++;
		if (V == Amount)
		{
```

```
                        lift(u); V = 0;
                }
        }
    }

    ~Flow()
    {
        delete[] e;
        delete[] h;
        for (int i = 0; i < Amount; ++i)
            delete[] c[i];
    }

    Flow(ifstream& file)
    {
        Map<char, int>* CharToNum = new Map<char, int>();
        Amount = 0;
        int StringN = 1;

        while (!file.eof())
        {
            string String;
            getline(file, String);
            if (String.size() >= 5) //cause five is min length of string to
correct input
            {
                if (!((String[0] >= 'A' && String[0] <= 'Z') && (String[1] ==
' ')))
                {
                    throw std::exception(string(("There is no space after
first symbol. Input is incorrect!   Line: " + to_string(StringN))).c_str());
                }
                if (!((String[2] >= 'A' && String[2] <= 'Z') && (String[3] ==
' ')))
                {
                    throw std::exception(string(("There is no space after
second symbol. Input is incorrect!   Line: " + to_string(StringN))).c_str());
                }
                string cur;

                for (int i = 4; i < String.size(); ++i)
                {
                    if (String[i] >= '0' && String[i] <= '9') cur +=
String[i];
                    else
                    {
                        throw std::exception(string(("There is some
trouble with third symbol. Input is incorrect!   Line: " + to_string(StringN))).c_str());
                    }
                }
                if (!CharToNum->find_is(String[0]))
                {
                    CharToNum->insert(String[0], Amount);
                    ++Amount;
                }
                if (!CharToNum->find_is(String[2]))
                {
                    CharToNum->insert(String[2], Amount);
                    ++Amount;
                }

            }
            else throw std::exception(string(("Data input is incorrect! Line: " +
to_string(StringN))).c_str());
            StringN++;
```

```
		}
//stok//
		if (CharToNum->find_is('S')) s = CharToNum->find('S');
		else throw std::exception("There is no Istok!");
//istok//
		if (CharToNum->find_is('T')) t = CharToNum->find('T');
		else throw std::exception("There is no Stok!");

		file.clear();
		file.seekg(ios::beg);
		e = new int[Amount];
		h = new int[Amount];
		c = new int*[Amount];
		for (int i = 0; i < Amount; ++i) {e[i] = 0;	h[i] = 0;}
		for (int i = 0; i < Amount; ++i)
		{
			c[i] = new int[Amount];
			for (int j = 0; j < Amount; ++j)
				c[i][j] = 0;
		}
		StringN = 1;
//itself//
		while (!file.eof())
		{
			string s1;
			int V1, V2;
			getline(file, s1);
			V1 = CharToNum->find(s1[0]);
			V2 = CharToNum->find(s1[2]);
			if (V1 == V2) throw std::exception(string("Vertex path to itself is
impossible! Line: " + to_string(StringN)).c_str());
			c[V1][V2] = stoi(s1.substr(4));
			StringN++;
		}
	}

private:
	int* e;
	int** c;
	int* h;
	int Amount, s, t;
};
```

List.h:

```
#pragma once
#include<iostream>
using namespace std;
template<class T>
class List
{
private:
	class Node {
	public:
		Node(T data = T(), Node* Next = NULL) {
			this->data = data;
			this->Next = Next;
		}
		Node* Next;
		T data;
	};
public:
	void push_back(T obj) { // добавление в конец списка bc
		if (head != NULL) {
```

```
                        this->tail->Next = new Node(obj);
                        tail = tail->Next;
                }
                else {
                        this->head = new Node(obj);
                        this->tail = this->head;
                }
                Size++;
        }
        void push_front(T obj) { // добавление в начало списка bc
                if (head != NULL) {
                        Node* current = new Node;
                        current->data = obj;
                        current->Next = this->head;
                        this->head = current;
                }
                else {
                        this->head = new Node(obj);
                        tail = head;
                }
                this->Size++;
        }
        void pop_back() { // удаление последнего элемента bc
                if (head != NULL) {
                        Node* current = head;
                        while (current->Next != tail)//то есть ищем предпоследний
                                current = current->Next;
                        delete tail;
                        tail = current;
                        tail->Next = NULL;
                        Size--;
                }
                else throw std::out_of_range("out_of_range");
        }
        void pop_front() { // удаление первого элемента bc-+
                if (head != NULL) {
                        Node* current = head;
                        head = head->Next;
                        delete current;
                        Size--;
                }
                else throw std::out_of_range("out_of_range");
        }
        void insert(T obj, size_t k) {// добавление элемента по индексу (вставка перед
элементом, который был ранее доступен по этому индексу) bc
                if (k >= 0 && this->Size > k) {
                        if (this->head != NULL) {
                                if (k == 0)
                                        this->push_front(obj);
                                else
                                        if (k == this->Size - 1)
                                                this->push_back(obj);
                                        else
                                        {
                                                Node* current = new Node;//для добавления
элемента
                                                Node* current1 = head;//для поиска итого
элемента
                                                for (int i = 0; i < k - 1; i++) {
                                                        current1 = current1->Next;
                                                }
                                                current->data = obj;
                                                current->Next = current1->Next;//переуказывает
на след элемент
                                                current1->Next = current;
```

```
                                        Size++;
                                }
                        }
                }
                else {
                        throw std::out_of_range("out_of_range");
                }
        }
        T at(size_t k) {// получение элемента по индексу bc
                if (this->head != NULL && k >= 0 && k <= this->Size - 1) {
                        if (k == 0)
                                return this->head->data;
                        else
                                if (k == this->Size - 1)
                                        return this->tail->data;
                                else
                                {
                                        Node* current = head;
                                        for (int i = 0; i < k; i++) {
                                                current = current->Next;
                                        }
                                        return current->data;
                                }
                }
                else {
                        throw std::out_of_range("out_of_range");
                }
        }
        void remove(int k) { // удаление элемента по индексу bc
                if (head != NULL && k >= 0 && k <= Size - 1) {
                        if (k == 0) this->pop_front();
                        else
                                if (k == this->Size - 1) this->pop_back();
                                else
                                        if (k != 0) {
                                                Node* current = head;
                                                for (int i = 0; i < k - 1; i++) {//переходим на
предэлемент
                                                        current = current->Next;
                                                }

                                                Node* current1 = current->Next;
                                                current->Next = current->Next->Next;
                                                delete current1;
                                                Size--;
                                        }
                }
                else {
                        throw std::out_of_range("out_of_range");
                }
        }
        size_t get_size() { // получение размера списка bc
                return Size;
        }
        void print_to_console() { // вывод элементов списка в консоль через разделитель,
не использовать at bc
                if (this->head != NULL) {
                        Node* current = head;
                        for (int i = 0; i < Size; i++) {
                                cout << current->data << ' ';
                                current = current->Next;
                        }
                }
        }
        void clear() { // удаление всех элементов списка
```

```
		if (head != NULL) {
			Node* current = head;
			while (head != NULL) {
				current = current->Next;
				delete head;
				head = current;
			}
			Size = 0;
		}
	}
	void set(size_t k, T obj)  // замена элемента по индексу на передаваемый элемент
	{
		if (this->head != NULL && this->get_size() >= k && k >= 0) {
			Node* current = head;
			for (int i = 0; i < k; i++) {
				current = current->Next;
			}
			current->data = obj;
		}
		else {
			throw std::out_of_range("out_of_range");
		}
	}
	bool isEmpty() { // проверка на пустоту списка bc
		return (bool)(head);
	}
	void reverse() { // меняет порядок элементов в списке
		int Counter = Size;
		Node* HeadCur = NULL;
		Node* TailCur = NULL;
		for (int j = 0; j < Size; j++) {
			if (HeadCur != NULL) {
				if (head != NULL && head->Next == NULL) {
					TailCur->Next = head;
					TailCur = head;
					head = NULL;
				}
				else {
					Node * cur = head;
					for (int i = 0; i < Counter - 2; i++)
						cur = cur->Next;
					TailCur->Next = cur->Next;
					TailCur = cur->Next;
					cur->Next = NULL;
					tail = cur;
					Counter--;
				}
			}
			else {
				HeadCur = tail;
				TailCur = tail;
				Node* cur = head;
				for (int i = 0; i < Size - 2; i++)
					cur = cur->Next;
				tail = cur;
				tail->Next = NULL;
				Counter--;
			}
		}
		head = HeadCur;
		tail = TailCur;
	}
public:
	List(Node* head = NULL, Node* tail = NULL, int Size = 0) :head(head), tail(tail),
Size(Size) {}
```

```
	~List() {
		if (head != NULL) {
			this->clear();
		}
	};
private:
	Node* head;
	Node* tail;
	int Size;
};
```

Map.h:

```
#pragma once
#define COLOR_RED 1
#define COLOR_BLACK 0
#include"List.h"
using namespace std;
template<typename T, typename T1>
class Map {
public:
	class Node
	{
	public:
		Node(bool color = COLOR_RED, T key = T(), Node* parent = NULL, Node* left =
NULL, Node* right = NULL, T1 value = T1()) :color(color), key(key), parent(parent),
left(left), right(right), value(value) {}
		T key;
		T1 value;
		bool color;
		Node* parent;
		Node* left;
		Node* right;
	};

	~Map()
	{
		if (this->Root != NULL)
			this->clear();
		Root = NULL;
		delete TNULL;
		TNULL = NULL;
	}

	Map(Node* Root = NULL, Node* TNULL = new Node(0)) :Root(TNULL), TNULL(TNULL) {}

	void printTree()
	{
		if (Root)
		{
			print_helper(this->Root, "", true);
		}
		else throw std::out_of_range("Tree is empty!");
	}

	void  insert(T key, T1 value)
	{
		if (this->Root != TNULL)
		{
			Node* node = NULL;
			Node* parent = NULL;
			/* Search leaf for new element */
			for (node = this->Root; node != TNULL; )
			{
```

```
                        parent = node;
                        if (key < node->key)
                                node = node->left;
                        else if (key > node->key)
                                node = node->right;
                        else if (key == node->key)
                                throw std::out_of_range("key is repeated");
                }

                node = new Node(COLOR_RED, key, TNULL, TNULL, TNULL, value);
                node->parent = parent;


                if (parent != TNULL)
                {
                        if (key < parent->key)
                                parent->left = node;
                        else
                                parent->right = node;
                }
                insert_fix(node);
        }
        else
        {
                this->Root = new Node(COLOR_BLACK, key, TNULL, TNULL, TNULL, value);
        }
}

List<T>* get_keys() {
        List<T>* list = new List<T>();
        this->ListKey(Root, list);
        return list;
}
List<T1>* get_values() {
        List<T1>* list = new List<T1>();
        this->ListValue(Root, list);
        return list;
}

T1 find(T key)
{
        Node* node = Root;
        while (node != TNULL && node->key != key)
        {
                if (node->key > key)
                        node = node->left;
                else
                        if (node->key < key)
                                node = node->right;
        }
        if (node != TNULL)
                return node->value;
        else
                throw std::out_of_range("Key is missing");
}

void remove(T key)
{
        this->delete_node(this->find_key(key));
}

void clear()
{
        this->clear_tree(this->Root);
        this->Root = NULL;
```

```
        }

        bool find_is(T key) {
                Node* node = Root;

                while (node != TNULL && node->key != key) {
                        if (node->key > key)
                                node = node->left;
                        else
                                if (node->key < key)
                                        node = node->right;
                }
                if (node != TNULL)
                        return true;
                else
                        return false;
        }
        void increment_value(T key) {
                Node* cur = this->find_value(key);
                cur->value++;
        }
private:
        Node* Root;
        Node* TNULL;

        //delete functions

        void delete_node(Node* find_node)
        {
                Node* node_with_fix, *cur_for_change;
                cur_for_change = find_node;
                bool cur_for_change_original_color = cur_for_change->color;
                if (find_node->left == TNULL)
                {
                        node_with_fix = find_node->right;
                        transplant(find_node, find_node->right);
                }
                else if (find_node->right == TNULL)
                {
                        node_with_fix = find_node->left;
                        transplant(find_node, find_node->left);
                }
                else
                {
                        cur_for_change = minimum(find_node->right);
                        cur_for_change_original_color = cur_for_change->color;
                        node_with_fix = cur_for_change->right;
                        if (cur_for_change->parent == find_node)
                        {
                                node_with_fix->parent = cur_for_change;
                        }
                        else
                        {
                                transplant(cur_for_change, cur_for_change->right);
                                cur_for_change->right = find_node->right;
                                cur_for_change->right->parent = cur_for_change;
                        }
                        transplant(find_node, cur_for_change);
                        cur_for_change->left = find_node->left;
                        cur_for_change->left->parent = cur_for_change;
                        cur_for_change->color = find_node->color;
                }
                delete find_node;
                if (cur_for_change_original_color == COLOR_RED)
                {
```

```
                this->delete_fix(node_with_fix);
            }
        }

        //swap links(parent and other) for rotate
        void transplant(Node* current, Node* current1)
        {
            if (current->parent == TNULL)
            {
                Root = current1;
            }
            else if (current == current->parent->left)
            {
                current->parent->left = current1;
            }
            else
            {
                current->parent->right = current1;
            }
            current1->parent = current->parent;
        }

        void clear_tree(Node* tree)
        {
            if (tree != TNULL)
            {
                clear_tree(tree->left);
                clear_tree(tree->right);
                delete tree;
            }
        }
    //find functions

        Node* minimum(Node* node)
        {
            while (node->left != TNULL)
            {
                node = node->left;
            }
            return node;
        }

        Node* maximum(Node* node)
        {
            while (node->right != TNULL)
            {
                node = node->right;
            }
            return node;
        }

        Node* grandparent(Node* current)
        {
            if ((current != TNULL) && (current->parent != TNULL))
                return current->parent->parent;
            else
                return TNULL;
        }

        Node* uncle(Node* current)
        {
            Node* current1 = grandparent(current);
            if (current1 == TNULL)
                return TNULL; // No grandparent means no uncle
            if (current->parent == current1->left)
```

```
                return current1->right;
        else
                return current1->left;
}

Node* sibling(Node* n)
{
        if (n == n->parent->left)
                return n->parent->right;
        else
                return n->parent->left;
}

Node* find_key(T key)
{
        Node* node = this->Root;
        while (node != TNULL && node->key != key)
        {
                if (node->key > key)
                        node = node->left;
                else
                        if (node->key < key)
                                node = node->right;
        }
        if (node != TNULL)
                return node;
        else
                throw std::out_of_range("Key is missing");
}

//all print function

void print_helper(Node* root, string indent, bool last)
{
        if (root != TNULL)
        {
                cout << indent;
                if (last)
                {
                        cout << "R----";
                        indent += "    ";
                }
                else
                {
                        cout << "L----";
                        indent += "|   ";
                }
                string sColor = !root->color ? "black" : "red";
                cout << root->key << " (" << sColor << ")" << endl;
                print_helper(root->left, indent, false);
                print_helper(root->right, indent, true);
        }
}

void list_key_or_value(int mode, List<T>* list)
{
        if (this->Root != TNULL)
                this->key_or_value(Root, list, mode);
        else
                throw std::out_of_range("Tree empty!");
}

void key_or_value(Node* tree, List<T>* list, int mode)
{
        if (tree != TNULL)
```

```
	{
		key_or_value(tree->left, list, mode);
		if (mode == 1)
			list->push_back(tree->key);
		else
			list->push_back(tree->value);
		key_or_value(tree->right, list, mode);
	}
}

//fix

void insert_fix(Node* node)
{
	Node* uncle;
	/* Current node is COLOR_RED */
	while (node != this->Root && node->parent->color == COLOR_RED)//
	{
		/* node in left tree of grandfather */
		if (node->parent == this->grandparent(node)->left)//
		{
			/* node in left tree of grandfather */
			uncle = this->uncle(node);
			if (uncle->color == COLOR_RED)
			{
				/* Case 1 - uncle is COLOR_RED */
				node->parent->color = COLOR_BLACK;
				uncle->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				node = this->grandparent(node);
			}
			else {
				/* Cases 2 & 3 - uncle is COLOR_BLACK */
				if (node == node->parent->right)
				{
					/*Reduce case 2 to case 3 */
					node = node->parent;
					this->left_rotate(node);
				}
				/* Case 3 */
				node->parent->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				this->right_rotate(this->grandparent(node));
			}
		}
		else {
			/* Node in right tree of grandfather */
			uncle = this->uncle(node);
			if (uncle->color == COLOR_RED)
			{
				/* Uncle is COLOR_RED */
				node->parent->color = COLOR_BLACK;
				uncle->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
				node = this->grandparent(node);
			}
			else {
				/* Uncle is COLOR_BLACK */
				if (node == node->parent->left)
				{
					node = node->parent;
					this->right_rotate(node);
				}
				node->parent->color = COLOR_BLACK;
				this->grandparent(node)->color = COLOR_RED;
```

```
                                this->left_rotate(this->grandparent(node));
                        }
                }
        }
        this->Root->color = COLOR_BLACK;
}

void delete_fix(Node* node)
{
        Node* sibling;
        while (node != this->Root && node->color == COLOR_BLACK)//
        {
                sibling = this->sibling(node);
                if (sibling != TNULL)
                {
                        if (node == node->parent->left)//
                        {
                                if (sibling->color == COLOR_BLACK)
                                {
                                        node->parent->color = COLOR_BLACK;
                                        sibling->color = COLOR_RED;
                                        this->left_rotate(node->parent);
                                        sibling = this->sibling(node);
                                }
                                if (sibling->left->color == COLOR_RED && sibling-
>right->color == COLOR_RED)
                                {
                                        sibling->color = COLOR_BLACK;
                                        node = node->parent;
                                }
                                else
                                {
                                        if (sibling->right->color == COLOR_RED)
                                        {
                                                sibling->left->color = COLOR_RED;
                                                sibling->color = COLOR_BLACK;
                                                this->left_rotate(sibling);
                                                sibling = this->sibling(node);
                                        }
                                        sibling->color = node->parent->color;
                                        node->parent->color = COLOR_RED;
                                        sibling->right->color = COLOR_RED;
                                        this->left_rotate(node->parent);
                                        node = this->Root;
                                }
                        }
                        else
                        {
                                if (sibling->color == COLOR_BLACK);
                                {
                                        sibling->color = COLOR_RED;
                                        node->parent->color = COLOR_BLACK;
                                        this->right_rotate(node->parent);
                                        sibling = this->sibling(node);
                                }
                                if (sibling->left->color == COLOR_RED && sibling-
>right->color)
                                {
                                        sibling->color = COLOR_BLACK;
                                        node = node->parent;
                                }
                                else
                                {
                                        if (sibling->left->color == COLOR_RED)
                                        {
```

```
                                        sibling->right->color = COLOR_RED;
                                        sibling->color = COLOR_BLACK;
                                        this->left_rotate(sibling);
                                        sibling = this->sibling(node);
                                }
                                sibling->color = node->parent->color;
                                node->parent->color = COLOR_RED;
                                sibling->left->color = COLOR_RED;
                                this->right_rotate(node->parent);
                                node = Root;
                        }
                }
        }

    }
    this->Root->color = COLOR_BLACK;
}
//Rotates

    void left_rotate(Node* node)
    {
        Node* right = node->right;
        /* Create node->right link */
        node->right = right->left;
        if (right->left != TNULL)
            right->left->parent = node;
        /* Create right->parent link */
        if (right != TNULL)
            right->parent = node->parent;
        if (node->parent != TNULL)
        {
            if (node == node->parent->left)
                node->parent->left = right;
            else
                node->parent->right = right;
        }
        else {
            this->Root = right;
        }
        right->left = node;
        if (node != TNULL)
            node->parent = right;
    }

    void right_rotate(Node* node)
    {
        Node* left = node->left;
        /* Create node->left link */
        node->left = left->right;
        if (left->right != TNULL)
            left->right->parent = node;
        /* Create left->parent link */
        if (left != TNULL)
            left->parent = node->parent;
        if (node->parent != TNULL)
        {
            if (node == node->parent->right)
                node->parent->right = left;
            else
                node->parent->left = left;
        }
        else
        {
            this->Root = left;
        }
```

```
			left->right = node;
			if (node != TNULL)
				node->parent = left;
		}
		void ListValue(Node* tree, List<T1>* list) {
			if (tree != TNULL) {
				ListValue(tree->left, list);
				list->push_back(tree->value);
				ListValue(tree->right, list);
			}
		}
		void ListKey(Node* tree, List<T>* list) {
			if (tree != TNULL) {
				ListKey(tree->left, list);
				list->push_back(tree->key);
				ListKey(tree->right, list);
			}
		}

		Node* find_value(T key) {
			Node* node = Root;

			while (node != TNULL && node->key != key) {
				if (node->key > key)
					node = node->left;
				else
					if (node->key < key)
						node = node->right;
			}
			if (node != TNULL)
				return node;

		}
};
```

Unit-тесты:

```
#include "pch.h"
#include "CppUnitTest.h"
#include <fstream>
#include "../Kurs/MaxFlow.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace KursTest
{
	TEST_CLASS(Exceptions)
	{
	public:
		TEST_METHOD(IncorrectFirst)
		{
			try
			{
				ifstream input("C:\\Kurs\\Input2.txt");
				Flow flow(input);
			}
			catch (exception & ex)
			{
				Assert::AreEqual(ex.what(), "There is no space after first
symbol. Input is incorrect!   Line: 2");
			}
		}
		TEST_METHOD(IncorrectSecond)
		{
```

```
			try
			{
				ifstream input("C:\\Kurs\\Input3.txt");
				Flow flow(input);
			}
			catch (exception & ex)
			{
				Assert::AreEqual(ex.what(), "There is no space after second
symbol. Input is incorrect!   Line: 2");
			}
		}
		TEST_METHOD(IncorrectThird)
		{
			try
			{
				ifstream input("C:\\Kurs\\Input4.txt");
				Flow flow(input);
			}
			catch (exception & ex)
			{
				Assert::AreEqual(ex.what(), "There is some trouble with third
symbol. Input is incorrect!   Line: 2");
			}
		}
		TEST_METHOD(Empty)
		{
			try
			{
				ifstream input("C:\\Kurs\\Input5.txt");
				Flow flow(input);
			}
			catch (exception & ex)
			{
				Assert::AreEqual(ex.what(), "Data input is incorrect! Line:
2");
			}
		}
		TEST_METHOD(VertexTotself)
		{
			try
			{
				ifstream input("C:\\Kurs\\Input8.txt");
				Flow flow(input);
			}
			catch (exception & ex)
			{
				Assert::AreEqual(ex.what(), "Vertex path to itself is
impossible! Line: 2");
			}
		}
	};

	TEST_CLASS(KursTest)
	{
	public:
		TEST_METHOD(TaskExample)
		{
			ifstream input("C:\\Kurs\\Input1.txt");
			Flow flow(input);
			Assert::AreEqual(flow.MaxFlow(), 5);
		}
		TEST_METHOD(TaskExampleIstokStok)
		{
			ifstream input("C:\\Kurs\\Input6.txt");
			Flow flow(input);
```

```
			Assert::AreEqual(flow.MaxFlow(), 25);
		}
		TEST_METHOD(IstokStokOnly)
		{
			ifstream input("C:\\Kurs\\Input7.txt");
			Flow flow(input);
			Assert::AreEqual(flow.MaxFlow(), 20);
		}
		TEST_METHOD(LargeExample)
		{
			ifstream input("C:\\Kurs\\Input9.txt");
			Flow flow(input);
			Assert::AreEqual(flow.MaxFlow(), 19);
		}
	};

}
```