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. 8302	_______________	Никулин Л.А.
Преподаватель	_______________	Тутуева А.В.

Санкт-Петербург
2020

1. Цель работы

Реализовать программу определяющую максимальный поток в заданном графе. Программа считывает из файла список ребер и их пропускные способности.

2. Описание реализуемого класса и методов

FlowPushRelabel	Содержит поля: int* excessFlowArray (массив избытков вершин), int** capacity (остаточная сеть), int* height (функция высоты), int vertexCount (количество вершин), int sourceVertex (исток), int destinationVertex (сток). Содержит следующие методы: Дефолтный конструктор. Конструктор с параметром ifstream&– вызывает метод setInfo(ifstream&); Деструктор – вызывает метод clear().
void push(int edge, int vertex)	Функция, проталкивающая поток из u в v, равный min{e[edge], cf(edge, vertex)}, и подсчитывающая остаточную сеть и избытки
void lift(int edge)	Поднимает вершину на минимальную высоту, достаточную для возможности проталкивания потока
void discharge(int edge)	Выполняет лифтинг и проталкивание, пока это возможно
int maximalFlow()	Вычисляет максимальный поток в сети
void clear()	Очищение на основе обычного удаления двоичного дерева
void setInfo(ifstream&)	Получает на вход файл с списком строк, обрабатывает их и выдает список смежности

3. Оценка временной сложности алгоритмов

void push(int edge, int vertex)	O(1)
void lift(int edge)	O(|V|)
void discharge(int edge)	O(|V| |E|)
int maximalFlow()	$O(\|V\|^2\|E\|)$

4. Описание реализованных unit-тестов

Test_Correct_output	Проверяет ситуацию с 20 вершинами
Test_Exception_entering_empty_character	Некорректное введение символа
Test_Exception_entering_the_bandwidth	Некорректное введение пропускной способности
Test_Exception_empty_string	Ввод пустой строки
Test_Exception_there_is_a_path_from_the_vertex_to_itself	Некорректный путь от вершины к самой себе

5. Обоснование выбора используемых структур данных

Был выбран MAP для того чтобы индивидуализировать вершины индексами. Данную структура используется потому, что она позволяет не сохранять повторяющиеся данные и быстрый доступ к ним. List используется для перебора вершин сети в функции maximalFlow.

Со структурой List удобнее работать, нежели с обычным массивом, так как не нужно хранить его размер, а также быстро добавлять и удалять элементы, без траты времени на их перезапись.

6. Примеры работы программы

№	Входные данные:	Результат:
1	S A 3 S C 2 S B 2 S D 1 A B 7 B F 5 C B 1 C E 6 D C 2 D E 2 E F 4 E I 1 F I 6 F G 3 G A 4 G H 7 H T 9 I T 7 I H 2	Максимальный поток: 8

2	S A 3 S C 3 S B 2 S D 1 A B 20 B F 5 C B 1 C E 6 D C 3 D E 2 E F 4 E I 1 F I 6 F G 3 G A 4 G H 7 H T 9 I T 10 I H 3	Максимальный поток: 9
3	S A 10 S B 10 A B 1 A C 8 A D 4 B D 5 B E 2 E D 5 C G 10 D G 2 G E 3 G F 1 E F 4 F T 3 D T 10 C H 8 H T 6	Максимальный поток: 17

4	S A 2 S B 10 S C 3 S D 2 A E 1 A F 1 B E 1 B H 1 C E 2 C H 2 D G 1 D H 1 E T 3 F T 2 G T 2 H T 3	Максимальный поток: 17
5	S A 2 S B 2 S C 2 A B 1 A D 2 D B 1 B C 1 C F 2 B F 2 A E 1 F E 1 E H 1 F G 1 F H 1 G H 1 E T 2 H T 2 G T 1 D T 1	Максимальный поток: 5

6	S A 10 S B 10 S C 5 A B 5 A F 5 B C 3 C D 5 D E 9 E H 5 D H 3 H F 6 H T 2 H G 2 G T 10 B F 2 B D 3 E T 4 F G 8	Максимальный поток: 15
7	S A 2 S B 10 S C 5 S D 5 A E 5 A F 5 B E 5 B H 1 C E 2 C H 2 D G 6 D H 16 E T 3 F T 2 G T 2 H T 3	Максимальный поток: 10

7. Листинг

CourseWork.cpp:

```
#include <iostream>
#include <fstream>
#include "Flow.h"

int main()
{
	ifstream input("input.txt");
	FlowPushRelabel example(input);
	std::cout << "Максимальный поток: " <<
example.maximalFlow();
}
```

Flow.h:

```
#pragma once
#include <fstream>
#include "List.h"
#include <string>
#include"Map.h"

using namespace std;
class FlowPushRelabel {
private:
	#pragma region VARIABLES
	int* excessFlowArray;
	int** capacity;
	int* height;
	int vertexCount, sourceVertex, destinationVertex;
	#pragma endregion

	#pragma region FUNCTIONS
	int min(int, int);
	#pragma endregion
public:
	FlowPushRelabel() = default;
	FlowPushRelabel(ifstream&);
	~FlowPushRelabel();

	int maximalFlow();
	void setInfo(ifstream&);
	void push(int, int);
```

```
	void clear();
	void lift(int);
	void discharge(int);
};
```

Flow.cpp:

```
#pragma once
#include "Flow.h"

FlowPushRelabel::~FlowPushRelabel() {
	clear();
}

FlowPushRelabel::FlowPushRelabel(ifstream& file)
{
	setInfo(file);
}

int FlowPushRelabel::maximalFlow() {
	if (vertexCount > 2) {
		for (int i = 0; i < vertexCount; i++)
		{
			if (i == sourceVertex)
				continue;
			excessFlowArray[i] = capacity[sourceVertex][i];
			capacity[i][sourceVertex] +=
capacity[sourceVertex][i];
		}
		height[sourceVertex] = vertexCount;
		List<int> l;
		int cur;
		int cur_index = 0;
		int old_height;
		for (int i = 0; i < vertexCount; i++)
			if (i != sourceVertex && i !=
destinationVertex)
				l.push_front(i);
		cur = l.at(0);
		while (cur_index < l.get_size())
		{
			old_height = height[cur];
			discharge(cur);
			if (height[cur] != old_height)
			{
```

```
                    l.push_front(cur);
                    l.remove(++cur_index);
                    cur = l.at(0);
                    cur_index = 0;
               }
               ++cur_index;
               if (cur_index < l.get_size())
                    cur = l.at(cur_index);

          }
          return excessFlowArray[destinationVertex];
     }
     else
          return capacity[0][1];
}

void FlowPushRelabel::setInfo(ifstream& file)
{
     Map<char, int>* cardCharNumber = new Map<char, int>();
     vertexCount = 0;
     int str_num = 1;
     while (!file.eof()) {
          string s1;
          getline(file, s1);
          if (s1.size() >= 5) {
               if (!((s1[0] >= 'A' && s1[0] <= 'Z') && (s1[1]
== ' ')) || !((s1[2] >= 'A' && s1[2] <= 'Z') && (s1[3] == '
'))) {
                    throw std::exception("Error entering a
character in the string or missing a space after it.");
               }
               string cur;
               for (int i = 4; i < s1.size(); ++i) {
                    if (s1[i] >= '0' && s1[i] <= '9')
                         cur += s1[i];
                    else {
                         throw std::exception("Error entering
the third character (bandwidth) in the string or the presence
of a space after it.");
                    }
               }
               if (!cardCharNumber->find_is(s1[0])) {
                    cardCharNumber->insert(s1[0],
vertexCount);
```

```
                    ++vertexCount;
               }
               if (!cardCharNumber->find_is(s1[2])) {
                    cardCharNumber->insert(s1[2],
vertexCount);
                    ++vertexCount;
               }

          }
          else
          {
               throw std::exception(string(("A data-entry
error. Check the correctness of the input in the file and
correct these errors in the line under the number: " +
to_string(str_num))).c_str());
          }
          ++str_num;
     }
     if (cardCharNumber->find_is('S'))
          sourceVertex = cardCharNumber->find('S');
     else {
          throw std::exception("Source is missing");
     }

     if (cardCharNumber->find_is('T'))
          destinationVertex = cardCharNumber->find('T');
     else {
          throw std::exception("Sink is missing");
     }
     file.clear();
     file.seekg(ios::beg);
     excessFlowArray = new int[vertexCount];
     height = new int[vertexCount];
     capacity = new int* [vertexCount];
     for (int i = 0; i < vertexCount; ++i) {
          excessFlowArray[i] = 0;
          height[i] = 0;
     }
     for (int i = 0; i < vertexCount; ++i) {
          capacity[i] = new int[vertexCount];
          for (int j = 0; j < vertexCount; ++j)
               capacity[i][j] = 0;
     }
     str_num = 1;
```

```
	while (!file.eof()) {
		string s1;
		int vert1, vert2, cap;
		getline(file, s1);
		vert1 = cardCharNumber->find(s1[0]);
		vert2 = cardCharNumber->find(s1[2]);
		if (vert1 == vert2)
			throw std::exception(string("The path from the
vertex to itself is impossible in the string under the number:
" + to_string(str_num)).c_str());
		capacity[vert1][vert2] = stoi(s1.substr(4));
		++str_num;
	}
}

void FlowPushRelabel::push(int edge, int vertex)
{
	int interVariable = min(excessFlowArray[edge],
capacity[edge][vertex]);
	excessFlowArray[edge] -= interVariable;
	excessFlowArray[vertex] += interVariable;
	capacity[edge][vertex] -= interVariable;
	capacity[vertex][edge] += interVariable;
}

void FlowPushRelabel::lift(int edge)
{
	int min = 2 * vertexCount + 1;

	for (int i = 0; i < vertexCount; i++)
		if (capacity[edge][i] && (height[i] < min))
			min = height[i];
	height[edge] = min + 1;
}

void FlowPushRelabel::clear()
{
	delete[] excessFlowArray;
	delete[] height;
	for (int i = 0; i < vertexCount; ++i)
	{
		delete[] capacity[i];
	}
}
```

```
void FlowPushRelabel::discharge(int edge)
{
	int vertex = 0;
	while (excessFlowArray[edge] > 0)
	{
		if (capacity[edge][vertex] && height[edge] ==
height[vertex] + 1)
		{
			push(edge, vertex);
			vertex = 0;
			continue;
		}
		++vertex;
		if (vertex == vertexCount)
		{
			lift(edge);
			vertex = 0;
		}
	}
}

int FlowPushRelabel::min(int data1, int data2) {
	return data1 > data2 ? data2 : data1;
}
```

Map.h:

```
#pragma once
#include"List.h"

using namespace std;

enum Color
{
	RED, BLACK
};

template<typename TypeKey, typename TypeValue>
class Map {
public:
	class Node
	{
	public:
```

```
          Node(bool color = RED, TypeKey key = TypeKey(),
Node* parent = NULL, Node* left = NULL, Node* right = NULL,
TypeValue value = TypeValue()) :color(color), key(key),
parent(parent), left(left), right(right), value(value) {}
          TypeKey key;
          TypeValue 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(TypeKey key, TypeValue 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(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(BLACK, key, TNULL, TNULL,
TNULL, value);
         }
    }

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

    TypeValue find(TypeKey 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(TypeKey key)
	{
		this->delete_node(this->find_key(key));
	}

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

	bool find_is(TypeKey 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(TypeKey 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 == 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(TypeKey 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<TypeKey>* 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<TypeKey>* 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 RED */
		while (node != this->Root && node->parent->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 == RED)
				{
					/* Case 1 - uncle is RED */
					node->parent->color = BLACK;
					uncle->color = BLACK;
					this->grandparent(node)->color = RED;
					node = this->grandparent(node);
				}
				else {
					/* Cases 2 & 3 - uncle is BLACK */
					if (node == node->parent->right)
					{
						/*Reduce case 2 to case 3 */
						node = node->parent;
```

```
                         this->left_rotate(node);
                    }
                    /* Case 3 */
                    node->parent->color = BLACK;
                    this->grandparent(node)->color = RED;
                    this->right_rotate(this-
>grandparent(node));
               }
          }
          else {
               /* Node in right tree of grandfather */
               uncle = this->uncle(node);
               if (uncle->color == RED)
               {
                    /* Uncle is RED */
                    node->parent->color = BLACK;
                    uncle->color = BLACK;
                    this->grandparent(node)->color = RED;
                    node = this->grandparent(node);
               }
               else {
                    /* Uncle is BLACK */
                    if (node == node->parent->left)
                    {
                         node = node->parent;
                         this->right_rotate(node);
                    }
                    node->parent->color = BLACK;
                    this->grandparent(node)->color = RED;
                    this->left_rotate(this-
>grandparent(node));
               }
          }
     }
     this->Root->color = BLACK;
}

void delete_fix(Node* node)
{
     Node* sibling;
     while (node != this->Root && node->color == BLACK)//
     {
          sibling = this->sibling(node);
          if (sibling != TNULL)
```

```
                {
                        if (node == node->parent->left)//
                        {
                                if (sibling->color == BLACK)
                                {
                                        node->parent->color = BLACK;
                                        sibling->color = RED;
                                        this->left_rotate(node->parent);
                                        sibling = this->sibling(node);
                                }
                                if (sibling->left->color == RED &&
sibling->right->color == RED)
                                {
                                        sibling->color = BLACK;
                                        node = node->parent;
                                }
                                else
                                {
                                        if (sibling->right->color ==
RED)

                                        {
                                                sibling->left->color = RED;
                                                sibling->color = BLACK;
                                                this->left_rotate(sibling);
                                                sibling = this-
>sibling(node);

                                        }
                                        sibling->color = node->parent-
>color;

                                        node->parent->color = RED;
                                        sibling->right->color = RED;
                                        this->left_rotate(node->parent);
                                        node = this->Root;
                                }
                        }
                        else
                        {
                                if (sibling->color == BLACK);
                                {
                                        sibling->color = RED;
                                        node->parent->color = BLACK;
                                        this->right_rotate(node-
>parent);

                                        sibling = this->sibling(node);
```

```
                    }
                    if (sibling->left->color == RED &&
sibling->right->color)
                    {
                         sibling->color = BLACK;
                         node = node->parent;
                    }
                    else
                    {
                         if (sibling->left->color == RED)
                         {
                              sibling->right->color =
RED;
                              sibling->color = BLACK;
                              this->left_rotate(sibling);
                              sibling = this-
>sibling(node);
                         }
                         sibling->color = node->parent-
>color;
                         node->parent->color = RED;
                         sibling->left->color = RED;
                         this->right_rotate(node-
>parent);
                         node = Root;
                    }
               }
          }

     }
     this->Root->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<TypeValue>* list) {
        if (tree != TNULL) {
            ListValue(tree->left, list);
```

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

        Node* find_value(TypeKey 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;

        }
};
```

List.h:

```
#pragma once
#include<iostream>
using namespace std;
template<class TypeKey>
class List
{
private:
        class Node {
        public:
            Node(TypeKey data = TypeKey(), Node* Next = NULL) {
                this->data = data;
                this->Next = Next;
            }
            Node* Next;
            TypeKey data;
```

```
    };
public:
    void push_back(TypeKey obj) { // добавление в конец
списка
        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(TypeKey obj) { // добавление в начало
списка
        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
        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
        if (head != NULL) {
            Node* current = head;
```

```
                head = head->Next;
                delete current;
                Size--;
            }
            else throw std::out_of_range("out_of_range");
        }
        void insert(TypeKey obj, size_t k) {// добавление
элемента по индексу (вставка перед элементом, который был
ранее доступен по этому индексу)
            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");
            }
        }
        TypeKey at(size_t k) {// получение элемента по индексу
            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
          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
          return Size;
     }
     void print_to_console() { // вывод элементов списка в
консоль через разделитель
```

```
        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, TypeKey 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
        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;
};
```

CourseWorkTests.cpp:

```
#include "CppUnitTest.h"
#include "../CourseWork/Flow.h"
#include "../CourseWork/Flow.cpp"
#include <fstream>
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

#define FILE1
"C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test1.
txt"
#define FILE2
"C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test2.
txt"
#define FILE3
"C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test3.
txt"
#define FILE4
"C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test4.
txt"
#define FILE5
"C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test5.
txt"
#define FILE6
"C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test6.
txt"

namespace UnitTestCourseWork
{
	TEST_CLASS(UnitTestCourseWork)
	{
	public:
		TEST_METHOD(Test_Correct_output)
		{
			ifstream stream(FILE1);
			FlowPushRelabel flow(stream);
			int excepted = 19;
			Assert::AreEqual(flow.maximalFlow(), excepted);
		}
		TEST_METHOD(Test_Exception_entering_empty_character)
{
			try {
				ifstream stream(FILE2);
				FlowPushRelabel flow(stream);
			}
```

```
                catch (exception& ex) {
                    Assert::AreEqual(ex.what(), "Error
entering a character in the string or missing a space after
it.");
                }
            }
            TEST_METHOD(Test_Exception_entering_the_bandwidth) {
                try {
                    ifstream stream(FILE3);
                    FlowPushRelabel flow(stream);
                }
                catch (exception& ex) {
                    Assert::AreEqual(ex.what(), "Error
entering the third character (bandwidth) in the string or the
presence of a space after it.");
                }
            }
            TEST_METHOD(Test_Exception_empty_string) {
                try {
                    ifstream stream(FILE4);
                    FlowPushRelabel flow(stream);
                }
                catch (exception& ex) {
                    Assert::AreEqual(ex.what(), "A data-entry
error. Check the correctness of the input in the file and
correct these errors in the line under the number: 2");
                }
            }

        TEST_METHOD(Test_Exception_there_is_a_path_from_the_verte
x_to_itself) {
                try {
                    ifstream stream(FILE6);
                    FlowPushRelabel flow(stream);
                }
                catch (exception& ex) {
                    Assert::AreEqual(ex.what(), "The path from
the vertex to itself is impossible in the string under the
number: 2");
                }
            }
        };
}
```