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 №2

Студент гр. 8301 Рыбакова Т.А.

Преподаватель Тутуева А.В.

Санкт-Петербург

2020

Цель работы

Реализовать алгоритм Эдмондса-Карпа для нахождения максимального потока в сети.

Описание реализуемого класса и методов

Описание классов и методов представлено в Таблице 1.

Таблица 1

Название	Назначение
Flow	Основной класс, в котором реализованы функции
void fillInd(string filename)	Записывает названия вершин и их индексы в мапы, затем передает число N-максимальный индекс вершины для выделения памяти для матриц.
void readInfo(string filename)	Считывает данные из файла
int findAugPath(int* parent)	выполняет поиск в ширину.
nt getMaxFlow()	выполняет алгоритм Эдмонса-Карпа, и возвращает максимальный поток через сеть.

Оценка временной сложности методов

Оценка временной сложности представлена в таблице №2

Таблица 2

unsigned getMaxFlow()	O(VE^2)
int findAugPath()	O(E)

Обоснование использованных структур данны

Для хранения пар название и индекс вершин был выбран ассоциативный массив. Реализация на основе красно-черного дерева.

Для хранения матрицы смежности, пропускных способностей и потоков ребер был выбрал динамический массив из-за простоты и скорости доступа.

Описание реализованных unit-тестов

Описание Unit-тестов представлено в Таблице 3.

Таблица 3

Название	Назначение
get_max_flow	обычный случай с правильным поведением
invalid_text	случай, когда в файле некорректно записана информация
duplicate_edge	случай, в файле дважды написана пропускная способность ребра
right_open_file	тест с правильными данными;
open_file_with_no_sink_source	тест с файлом, в котором отсутствует источник и сток;
negative_capacity	тест с файлом, имеющим отрицательное значение на ребре;

Пример работы программы

```
S A 20
S B 30
S C 20
A D 20
A F 5
B A 5
B E 20
C B 5
C F 20
F B 10
F H 20
F I 10
E A 5
E D 5
E F 5
E G 5
E H 20
D G 20
D I 20
G H 5
G T 20
H T 30
H I 5
I T 25
```

```
S A 4
S B 5
S C 10
S D 5
A F 2
A E 4
E F 1
E G 2
E H 7
E I 2
E T 10
B E 15
C E 11
D E 12
D I 8
F G 2
G T 5
I H 5
H T 9
```

```
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
```

```
S A 2
S B 2
S C 1
S D 1
A E 2
E B 1
B F 2
F C 1
C G 2
G D 1
D H 2
E T 2
F T 2
G T 2
H T 2
```

```
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
```

```
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
```

```
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
```

Консоль отладки Microsoft Visual Studio

```
MaxFlow: 65
MaxFlow: 23
MaxFlow: 6
MaxFlow: 17
MaxFlow: 8
MaxFlow: 5
MaxFlow: 15
```

Листинг

TanyaRybakova.cpp:

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

using namespace std;

void printMaxFlow(string &file)
{
	Flow flow(file);

	cout << "MaxFlow: " << flow.getMaxFlow() << endl;
}

int main()
{

	for (int i = 1; i < 8; ++i)
	{
		string fileName = "INPUT" + to_string(i) + ".txt";

		printMaxFlow(fileName);
	}

	return 0;
}
```

Flow.h:

```
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <exception>
#include "Map.h"
using namespace std;
class Flow {
private:
	int** edge, **capacity, **adj;
	int indS, indT, maxFlowFormat, maxFlow, N;
	void readInfo(string);
	void fillInd(string);
	int findAugPath(int*);
	Map<string, int> indices;
	Map<int, string> names;
public:
	Flow(string);
```

```
    ~Flow();
    int getMaxFlow();
};

Flow::~Flow()
{
    for (size_t i = 0; i < N; i++)
    {
        delete[] adj[i];
        delete[] capacity[i];
        delete[] edge[i];
    }
}

Flow::Flow(string filename)
{
    maxFlowFormat = INT_MIN;
    maxFlow = -1;
    fillInd(filename);
    edge = new int* [N];
    adj = new int* [N];
    capacity = new int* [N];
    for (int i = 0; i < N; i++) {
        edge[i] = new int[N];
        adj[i] = new int[N];
        capacity[i] = new int[N];
        for (int j = 0; j < N; j++) {
            adj[i][j] = 0;
            capacity[i][j] = 0;
            edge[i][j] = 0;
        }
    }
    readInfo(filename);
}

int Flow::getMaxFlow() {
    if (this->maxFlow != -1)
    {
        return maxFlow;
    }
    int flow = 0;
    int* parent = new int[N];
    int aug_flow = findAugPath(parent);
    while (aug_flow) {
        flow += aug_flow;
        int current = indT;
        while (current != indS) {
            int prev = parent[current];
            capacity[prev][current] -= aug_flow;
```

```
                capacity[current][prev] += aug_flow;
                edge[prev][current] += aug_flow;
                maxFlowFormat = edge[prev][current] >
maxFlowFormat ? edge[prev][current] : maxFlowFormat;
                current = prev;
            }
            aug_flow = findAugPath(parent);
        }
        this->maxFlow = flow;
        return flow;
}

void Flow::readInfo(string filename) {
        ifstream file(filename);
        if (!file.good())
            throw exception("Can't read this file");
        for (string line; getline(file, line);) {
            istringstream iss(line);
            string u, v, c_s;
            getline(iss, u, ' ');
            getline(iss, v, ' ');
            getline(iss, c_s, ' ');
            int c = std::stoi(c_s);
            if (c <= 0)
                throw exception("Capacity can only be a positive
integer");
            auto u_i = indices.find(u);
            auto v_i = indices.find(v);
            if (capacity[u_i][v_i] != 0)
                throw exception("File contains two lines
describing one edge");
            capacity[u_i][v_i] = c;
            adj[u_i][v_i] = v_i;
            adj[v_i][u_i] = u_i;
        }
        file.close();
}

void Flow::fillInd(string filename) {
        ifstream file(filename);
        if (!file.good())
            throw exception("Can't read this file");
        int max_index = 0;
        for (string line; getline(file, line);) {
            istringstream iss(line);
            string u, v, c_s;
            getline(iss, u, ' ');
            getline(iss, v, ' ');
            getline(iss, c_s, ' ');
```

```
		try {
			indices[u];
		}
		catch (...) {
			indices.insert(u, max_index);
			names.insert(max_index, u);
			if (u == "S")
				indS = max_index;
			else if (u == "T")
				indT = max_index;
			max_index++;
		}
		try {
			indices[v];
		}
		catch (...) {
			indices.insert(v, max_index);
			names.insert(max_index, v);
			if (v == "S")
				indS = max_index;
			else if (v == "T")
				indT = max_index;
			max_index++;
		}
	}
	file.close();
	try {
		indices["S"];
		indices["T"];
	}
	catch (...) {
		throw exception("File does not contain a source or
sink (or both)");
	}
	N = max_index;
}

int Flow::findAugPath(int* parent) {
	for (size_t i = 0; i < N; i++)
		parent[i] = -1;
	Queue<pair<int, int>> queue;
	queue.enqueue(make_pair(indS, INT_MAX));
	while (!queue.isEmpty()) {
		int current = queue.front().first;
		int flow = queue.front().second;
		queue.dequeue();
		for (int i = 0; i < N; i++) {
			int next = adj[current][i];
			if (next == -1)
```

```
                    continue;
                if (parent[next] == -1 && capacity[current][next]
!= 0) {
                    parent[next] = current;
                    int aug_flow = flow <
capacity[current][next] ? flow : capacity[current][next];
                    if (next == indT)
                        return aug_flow;
                    queue.enqueue({ next, aug_flow });
                }
            }
        }
    return 0;
}
```

Map.h:

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


enum Color {
    Black, Red, DoubleBlack
};

template<typename TKey, typename
TValue> class Map {
private:
    class Node {
    public:
        TKey key;
        TValue
        value;
        Color color;
        Node *parent, *left, *right;

        Node() = default;

        // Default color is Red, node isn't linked
        explicit Node(TKey key, TValue val, Node *nil) :
key(key), value(val), color(Color::Red),

parent(nullptr), left(nil), right(nil) {};

        // Constructor with parent and color specified
        Node(TKey key, TValue val, Node *nil, Color color) :
key(key), value(val), color(color),
```

```
parent(nullptr), left(nil), right(nil) {};

        // --------- Auxiliary methods -----------------
```

```
        Node *get_grandpa() {
            auto parent = this->parent;
            if (!parent || !parent->parent)
                return nullptr;
            auto grandpa = parent->parent;
            return grandpa;
        }

                                Node *get_uncle() {
                                    auto grandpa =
                                    this->get_grandpa();
                                    // If nodes parent is a
child                               right child, return left

                                    if
                                        (gran
                                        dpa->
                                        right
                                        ==
                                        this-
                                        >pare
                                        nt)
                                        retur
                                        n
                                        grand
                                        pa->l
                                        eft;
                                    else
                                        return grandpa->right;

        }
        // ---------------------------------------------
    };

    void clear(Node *node) {
        if (node == nullptr || node ==
            nil) return;
        clear(node->left);
        clear(node->right)
        ; delete node;
        node = nil;
        if (node->left !=
            nil) node->left =
            nil;
        if (node->right !=
            nil) node->right =
            nil;
```

```
    }

    Node *bst_find(const TKey &key, bool insertion = false) {
        auto node = this->root;
        auto prev = node;
        while (node && node != this->nil && node->key != key)
{
            prev = node;
            if (key > node->key)
                node = node->right;
            else
                node = node->left;
        }
        // If node with this key exist, return it
(deletion use-case)
        if (node != this->nil) {
            if (!insertion) {
```

```
            return node;
        } else {
            if (node && node->key == key)
                throw std::out_of_range("Duplicate key");
            else
                throw std::out_of_range("Invalid key");
        }
    } else {
        if (insertion)
            return prev;
        else
            throw std::out_of_range("Invalid key");
    }
}

Node *bst_successor(Node *node) {
    node = node->right;
    while (node->left !=
        nil) node =
        node->left;
    return node;
}

Node *left_rotation(Node *node) {
    auto new_node = new Node(node->key, node->value, nil);

    if (node->left != nil && node->right->left !=
        nil) new_node->right = node->right->left;
    new_node->left = node->left;
    new_node->color =
    node->color;

    node->key = node->right->key;
    node->value =
    node->right->value; node->left =
    new_node;

    if (new_node->left)
        new_node->left->parent = new_node;
    if (new_node->right)
        new_node->right->parent =
    new_node; new_node->parent = node;

    if (node->right &&
        node->right->right) node->right =
        node->right->right;
    else
```

```
    node->right = nil;

if (node->right)
    node->right->parent = node;
return new_node;
```

```
    }

    Node *right_rotation(Node *node) {
        auto new_node = new Node(node->key, node->value, nil);

        if (node->left != nil && node->left->right !=
            nil) new_node->left = node->left->right;
        new_node->right =
        node->right; new_node->color
        = node->color;

        node->key = node->left->key;
        node->value =
        node->left->value;

        node->right = new_node;
        if (new_node->left)
            new_node->left->parent = new_node;
        if (new_node->right)
            new_node->right->parent =
        new_node; new_node->parent = node;


        if (node->left &&
            node->left->left) node->left =
            node->left->left;
        else
            node->left = nil;

        if (node->left)
            node->left->parent = node;
        return new_node;
    }

    void resolve_insert_violations(Node *node) {
        while (node->parent->color == Color::Red &&
node->color == Color::Red) {
            auto grandpa = node->get_grandpa();
            auto uncle = node->get_uncle();
            // If parent is a left child of grandparent
            if (grandpa->left == node->parent) {
                if (uncle->color == Color::Red) {
                    node->parent->color =
                    Color::Black; uncle->color =
                    Color::Black; grandpa->color =
                    Color::Red;
                    if (grandpa != root)
```

```
        node = grandpa;
    else
        break;
} else if (node == grandpa->left->right) {
```

```
                    // If uncle's color is black or it's null
and path is LEFT-RIGHT
                    node = left_rotation(node->parent);
                } else {
                    grandpa->color = Color::Red;
                    node = right_rotation(grandpa);
                    node->parent->color = Color::Black;
                    if (grandpa != root)
                        node = grandpa;
                    else
                        break;
                }
            } else {
                // If parent is a right child of grandparent
                if (uncle->color == Color::Red) {
                    node->parent->color =
                    Color::Black; uncle->color =
                    Color::Black; grandpa->color =
                    Color::Red;
                    if (grandpa != root)
                        node = grandpa;
                    else
                        break;
                } else if (node == grandpa->right->left)
                    node = right_rotation(node->parent);
                else {
                    grandpa->color = Color::Red;
                    node = left_rotation(grandpa);
                    node->parent->color = Color::Black;
                    if (grandpa != root)
                        node = grandpa;
                    else
                        break;
                }
            }
        }
        root->color = Color::Black;
    }

    void remove_with_fix(Node *node) {
        if (node == root) {
            delete root;
            root = nullptr;
            return;
        }
```

```
        // Simple case
        if (node->color == Color::Red || node->left->color ==
Color::Red || node->right->color == Color::Red) {
```

```
                                // Choose left child if it
                                exists, else choose
right child
                                auto child = node->left != nil
                                ? node->left :
node->right;

            // Remove node
            if (node == node->parent->left) {
                node->parent->left = child;
                if (child != nil)
                    child->parent =
                node->parent; child->color =
                Color::Black; delete node;
            } else {
                node->parent->right = child;
                if (child != nil)
                    child->parent =
                node->parent; child->color =
                Color::Black; delete node;
            }
        }
            // Cases with black node
        else {
            Node *sibling = nullptr;
            Node *parent = nullptr;
            Node *ptr = node;
            ptr->color = Color::DoubleBlack;

            while (ptr != root && ptr->color ==
Color::DoubleBlack) {
                parent = ptr->parent;
                // If double-black node is a left child
                if (ptr == parent->left) {
                    sibling = parent->right;
                    // If sibling's color is red
                    if (sibling->color == Color::Red)
                        { sibling->color =
                        Color::Black; parent->color =
                        Color::Red;
                        left_rotation(parent);
                    }
                        // If sibling's color is black
                    else {
                        if (sibling->left->color ==
Color::Black && sibling->right->color == Color::Black) {
```

```
sibling->color = Color::Red;
if (parent->color == Color::Red)
    parent->color =
    Color::Black;
else
```

```
                                            parent->color =
Color::DoubleBlack;
                                        ptr = parent;
                                    } else {
                                        if
Color::Black) { Color::Black;
                                            (sibling->r
                                            ight->color
                                            ==
                                            sibling->le
Color::Black;
                                            ft->color =
                                            sibling->color
                                            = Color::Red;
                                            right_rotation(
                            }               sibling);
                                            sibling =
                                            parent->right;
                                        }
                                        sibling->color =
                                        parent->color;
                                        parent->color =
                                        Color::Black;
                                        sibling->right->co
                                        lor =
                                        left_rota
                                        tion(pare
                                        nt);
                                        break;
                                    }
                                }

                    // If double-black node is a right child
                else {
                    sibling = parent->left;
                    // If sibling's color is red
                    if (sibling->color == Color::Red)
                        { sibling->color =
                        Color::Black; parent->color =
                        Color::Red;
                        right_rotation(parent);
                    }
```

```
// If sibling's color is black

else {
    // and both its
recolor children are black,

    if
    (sibling->left->color
    ==

Color::Black && sibling->right->color == Color::Black) {
    sibling->color = Color::Red;
    if (parent->color == Color::Red)
        parent->color =
        Color::Black;
    else
        parent->color =

Color::DoubleBlack;        ptr = parent;

    }   // else, when at least one of
        its
children is red, rotate

else {
```

```
                                                    if
Color::Black) { Color::Black;
                                                        (sibling-
                                                        >left->co
                                                        lor ==
                                                        sibling->
Color::Black;
                                                        right->co
                                                        lor =
                                                        sibling->color
                }                                       = Color::Red;
            }                                           left_rotation(
                                                        sibling);
                                                        sibling =
                                                        parent->left;
                                                    }
                                                    sibling->color =
                                                    parent->color;
                                                    parent->color =
                                                    Color::Black;
                                                    sibling->left->co
                                                    lor =

                                                    right_rot
                                                    ation(par
                                                    ent);
                                                    break;
                                                }
                                            }

            if (node ==
                node->parent->left)
                node->parent->left = nil;
            else
                node->parent->right = nil;
            delete node;
            root->color = Color::Black;
        }
    }

// Map private fields
    Node *root;
```

```
    Node *nil;

public:

    Map() : root(nullptr), nil(new Node())
        { nil->left = nil;
        nil->right = nil;
    };


    void insert(const TKey &key, TValue value) {
        auto new_node = new Node(key, value, nil);
        // If tree is empty
        if (root == nullptr)
            { root = new_node;
            root->color = Color::Black;
        } else {
            // Find a place for a node violating RB-tree
properties
```

```
        auto parent = this->bst_find(key, true);
        if (new_node->key > parent->key)
            parent->right = new_node;
        else
            parent->left = new_node;
        new_node->parent = parent;
        resolve_insert_violations(new_node);
    }
}

void remove(const TKey &key) {
    // Find node to remove
    auto node = this->bst_find(key);
    if (node == nullptr || node ==
        nil) return;

    if (node->left != nil && node->right == nil) {
        // Case when node has only left
        child node->key = node->left->key;
        node->value = node->left->value;
        // Select child for removal
        node = node->left;
    } else if (node->right != nil && node->left == nil) {
        // Case when node has only right child
        node->key = node->right->key;
        node->value = node->right->value;
        // Select child for removal
        node = node->right;
    } else if (node->left != nil && node->right != nil) {
        // Case when node has both children
        auto successor = this->bst_successor(node);
        node->key = successor->key;
        node->value = successor->value;
        node = successor;
    }
    remove_with_fix(node);
}

void set(const TKey &key, TValue value) {
    auto node = bst_find(key);
    node->value = value;
}

TValue  find(const  TKey  &key)
    {     auto     node     =
    bst_find(key);  if  (node
```

```
== nullptr)
    throw     std::out_of_range("Key     doesn't
exist"); if (node->key == key)
```

```
        return node->value;
    else {
        if (node->left == node) {
            if (node->left->key == key)
                return node->left->value;
        } else {
            if (node->right->key == key)
                return node->right->value;
        }
    }
}

TValue operator[](const TKey &key) {
    return find(key);
}

void clear() {
    clear(root);
    root =
    nullptr;
}

List<TKey> get_keys() {
    List<TKey> keys_list;
    if (root) {
        auto node = root;
        List<Node *> stack;
        stack.push_front(node);
        while (!stack.isEmpty())
        {
            node = stack.at(0);
            stack.pop_front();
            keys_list.push_back(node->key);
            if (node->right != nil)
                stack.push_front(node->right);
            if (node->left != nil)
                stack.push_front(node->left);
        }
    }
    return keys_list;
}

List<TValue> get_values() {
    List<TValue>
    values_list; if (root) {
        auto node = root;
        List<Node *> stack;
```

```
stack.push_front(node);
while (!stack.isEmpty())
{
    node = stack.at(0);
```

```
                stack.pop_front();
                values_list.push_back(node->value)
                ; if (node->right != nil)
                    stack.push_front(node->right);
                if (node->left != nil)
                    stack.push_front(node->left);
            }
        }
        return values_list;
    }

    void print() {
        std::cout << '{';
        if (root) {
            auto node = root;
            List<Node *> stack;
            stack.push_front(node);
            while (!stack.isEmpty())
            {
                node = stack.at(0);
                if (node != root)
                    std::cout << ", ";
                stack.pop_front();
                std::cout << node->key << ": " << node->value;
                if (node->right != nil)
                    stack.push_front(node->right);
                if (node->left != nil)
                    stack.push_front(node->left);
            }
        }
        std::cout << '}' << std::endl;
    }
};
```

List.h:

```
#ifndef LIST_H
#define LIST_H
#include <iostream>

template<typename
T> class List {
private:
    // Defining nested class describing node
    class Node {
    public:
```

```
T data;
Node *prev, *next;
```

```
        // A constructor for Node class
        explicit Node(T value) : data(value), prev(nullptr),
next(nullptr) {}
        Node(T value, Node* prev, Node* next) : data(value),
prev(prev), next(next) {}
    };

    size_t size;
    Node *head, *tail;
public:
    // A default constructor for List class
    List() : head(nullptr), tail(nullptr), size(0) {}

    ~List() {
        this->clear();
    }

    void push_back(T value) {
        Node *temp = new Node(value);
        if (!isEmpty()) {
            temp->prev = tail;
            tail->next = temp;
            tail = temp;
            size++;
        } else {
            head       =
            temp;   tail
            =      temp;
            size++;
        }
    }

    void push_front(T value) {
        Node      *temp      =      new
        Node(value);                 if
        (!isEmpty()){
        head->prev       =
        temp;   temp->next
        =  head;  head   =
        temp;
            size++;
    } else {
        head = temp;
        tail = temp;
            size++;
    }
```

```
 }

void pop_back() {
     Node *temp = tail;
     if (head != tail) {
```

```
        tail = tail->prev;
        tail->next =
        nullptr; delete
        temp;
        size--;
    } else {
        if
            (!isEmpty()
            ) size--;
        head          =
        nullptr;   tail
        =      nullptr;
        delete temp;
    }
}

void pop_front() {
    Node *temp =
    head;
    if (head != tail) {
        head = head->next;
        head->prev = nullptr;
        delete temp;
        size--;
    } else {
        if
            (!isEmpty()
            ) size--;
        head          =
        nullptr;   tail
        =      nullptr;
        delete temp;
    }
}
```

pos

```
void
insert(T value, size_t pos) {
if (pos >= size || pos < 0)
throw std::out_of_range("Out of range exception");
size_t i = 0;
Node* insertion_node = head;
if (pos == 0)
push_fron
```

```
t(value);
else if (pos == size - 1)
    push_back(value);
else {
    // Iterate through list to the node before one at

    while (i < pos - 1) {
        insertion_node = insertion_node->next; i++;
    }
    // Insert new node at pos
    Node *temp = new Node(value,
insertion_node, insertion_node->next);
```

```
            insertion_node->next = temp;
            size++;
        }
    }

    T at(size_t pos) {
        if (pos >= size || pos < 0)
            throw std::out_of_range("Out of range exception");
        size_t i = 0;
        Node *temp = head;
        // Iterate through list to the node at pos
        while (i < pos) {
            temp = temp->next;
            i++;
        }
        return temp->data;
    }

    void remove(size_t pos) {
        if (pos >= size || pos < 0)
            throw std::out_of_range("Out of range exception");
        size_t i = 0;
        Node *temp = head;
        // Iterate through list to the node at pos
        while (i < pos) {
            temp = temp->next;
            i++;
        }
        if (temp == head) {
            if (!isEmpty())
                size--;
            head = head->next;
            head->prev =
            nullptr; delete
            temp;
        } else if (temp == tail) {
            if (!isEmpty())
                size--;
            tail = tail->prev;
            tail->next =
            nullptr; delete
            temp;
        } else {
                size--;
            temp->prev->next             =
            temp->next;   temp->next->prev
```

```
        = temp->prev; delete temp;
    }
}
```

```
    size_t get_size() {
        return size;
    }

    void print_to_console() {
        Node *temp = head;
        while (temp) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }

    void clear() {
        while (head)
            pop_front();
        size = 0;
    }

    void set(size_t pos, T value) {
        if (pos >= size || pos < 0)
            throw std::out_of_range("Out of range exception");
        size_t i = 0;
        Node *temp = head;
        // Iterate through list to the node at pos
        while (i < pos) {
            temp = temp->next;
            i++;
        }
        // If trying to access list at not existing
index, throw an exception
        temp->data = value;
    }

    size_t find_first(List sublist) {
        const size_t list_s = this->get_size();
        const size_t sublist_s =
        sublist.get_size();
        // If sublist is longer than list to be searched,
return error code
        if (sublist_s > list_s || sublist.isEmpty()
|| this->isEmpty())
            return -1;
        size_t i = 0;  // Index of first appearance of
        sublist Node *node = head;
        // Iterate through list excluding last sublist_s - 1
```

```
elements
        while (i <= list_s - sublist_s) {
```

```
                                  Node *temp = node;  //
list sublist
                                  Creating temp node of main

                                  size_t sublist_i = 0;  //

                                  Creating counter for

                                  while (sublist_i < sublist_s
                                  && temp->data ==

sublist.at(sublist_i)) {
                temp = temp->next;
                sublist_i++;
            }
            if (sublist_i == sublist_s)
                return i;
            i++;
            node = node->next;
        }
        return -1;
    }

    bool isEmpty() {
        return this->head == nullptr;
    }
};

template<typename
T> class Queue {
private:
    // Defining nested class describing node
    class Node {
    public:
        T data;
        Node *prev, *next;

        // A constructor for Node class
        explicit Node(T value) : data(value), prev(nullptr),
next(nullptr) {}
        Node(T value, Node* prev, Node* next) : data(value),
prev(prev), next(next) {}
    };

    size_t size;
    Node *head, *tail;
public:
```

```
// A default constructor for List class
Queue() : head(nullptr), tail(nullptr), size(0) {}

~Queue() {
    this->clear();
}
```

```
void enqueue(T value) {
    Node *temp = new Node(value);
    if (!isEmpty()) {
        temp->prev = tail;
        tail->next = temp;
        tail = temp;
        size++;
    } else {
        head       =
        temp;   tail
        =      temp;
        size++;
    }
}

void dequeue() {
    Node *temp = head;
    if (head != tail) {
        head = head->next;
        head->prev =
        nullptr; delete
        temp;
        size--;
    } else {
        if
            (!isEmpty()
            ) size--;
        head          =
        nullptr;   tail
        =      nullptr;
        delete temp;
    }
}

T front(){
    if (this->head == nullptr)
        throw std::out_of_range("Queue is empty");
    return this->head->data;
}

size_t get_size() {
    return size;
}

bool isEmpty() {
    return this->head == nullptr;
```

```
}

void clear() {
    while (head)
        dequeue();
    size = 0;
```

```
    }
};
#endif
```

Tests.cpp:

```
#include "CppUnitTest.h"
#include <stdexcept>
#include "../TanyaRybakova/Flow.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;
using namespace std;

namespace Tests
{
    TEST_CLASS(Tests)
    {
    public:

        TEST_METHOD(right_open_file)
        {
            try
            {
                Flow testFlow("EX1.txt");
            }
            catch (const exception& ex) {
                Assert::IsTrue(false);
            }
        }
        TEST_METHOD(get_max_flow)
        {
            Flow testFlow("ex1.txt");
            int excepted = 65;
            Assert::AreEqual(excepted,
testFlow.getMaxFlow());
        }
        TEST_METHOD(negative_capacity)
        {
            try
            {
                Flow testFlow("NEGATIVE.txt");
            }
```

```
                catch (const exception& ex) {
                    Assert::AreEqual(ex.what(), "Capacity can
only be a positive integer");
                }
            }
            TEST_METHOD(duplicate_edge)
            {
                try
                {
                    Flow testFlow("DUP.txt");
                }
                catch (const exception& ex) {

                    Assert::AreEqual(ex.what(), "File contains
two lines describing one edge");
                }
            }
            TEST_METHOD(invalid_text)
            {
                try
                {
                    Flow testFlow("INVALID.txt");
                }
                catch (const exception& ex) {

                    Assert::AreEqual(ex.what(), "invalid stoi
argument");
                }
            }
            TEST_METHOD(no_sink_source)
            {
                try
                {
                    Flow testFlow("NOSINK.txt");
                }
                catch (const exception& ex) {
                    Assert::AreEqual(ex.what(), "File does not
contain a source or sink (or both)");
                }
            }
        };
}
```