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»

Тема: Потоки в сетях

Студент гр. 0302 _____ Касаткин А.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2022

Задача: Найти максимальный поток, используя алгоритм Эдмондса-Карпа

Описание алгоритма:

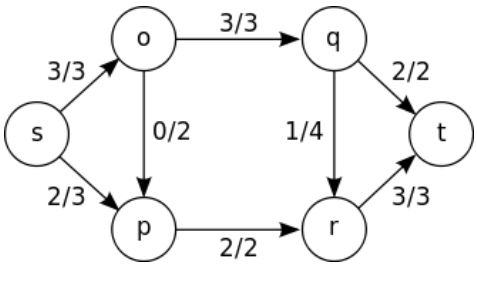
1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим кратчайший путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь (он называется **увеличивающим путём** или **увеличивающей цепью**) максимально возможный поток:
    1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью c_{min}.
    2. Для каждого ребра на найденном пути увеличиваем поток на c_{min}, а в противоположном ему — уменьшаем на c_{min}.
    3. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4. Возвращаемся на шаг 2.

Чтобы найти кратчайший путь в графе, используем поиск в ширину:

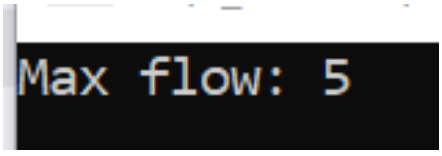
1. Создаём очередь вершин *O*. Вначале *O* состоит из единственной вершины *s*.
2. Отмечаем вершину *s* как посещённую, без родителя, а все остальные как не посещённые.
3. Пока очередь не пуста, выполняем следующие шаги:
    1. Удаляем первую в очереди вершину *u*.
    2. Для всех дуг (*u*, *v*), исходящих из вершины *u*, для которых вершина *v* ещё не посещена, выполняем следующие шаги:
        1. Отмечаем вершину *v* как посещённую, с родителем *u*.
        2. Добавляем вершину *v* в конец очереди.
        3. Если *v* = *t*, выходим из обоих циклов: мы нашли кратчайший путь.
4. Если очередь пуста, возвращаем ответ, что пути нет вообще и останавливаемся.
5. Если нет, идём от *t* к *s*, каждый раз переходя к родителю. Возвращаем путь в обратном порядке.

Оценка временной сложности:
Maxflow O(VE)

Результат:



S O 3
S P 3
O Q 3
O P 2
P R 2
Q R 4
Q T 2
R T 3



Листинг:

Main.cpp

```
#include "Karp.h"
#include <fstream>

int main() {
	Matrix<int> M;
	Matrix<int> adj;
	List<char> V;

	{
		ifstream file("input.txt");
		if (!file.is_open())
			throw runtime_error("File was not found");
		int symb_num = 0;
		char symb;
		int cap;
		char v1, v2;
		while (!file.eof()) {
			file >> noskipws >> symb;
			if ((symb_num == 0 || symb_num == 2) && !V.contains(symb))
				V.push_back(symb);
			symb_num++;
			if (symb == '\n' || file.eof())
				symb_num = 0;
		}

		if (V.at(0) != 'S')
			throw invalid_argument("Graph should start with 'S'-vertex");

		if (V.at(V.get_size() - 1) != 'T')
			throw invalid_argument("Graph should finish with 'T'-vertex");

		file.clear();
		file.seekg(0);

		List<int> temp_list;

		for (int i = 0; i < V.get_size(); i++) {
			for (int j = 0; j < V.get_size(); j++)
				temp_list.push_back(0);
			M.push_back(temp_list);
			temp_list.reset();
		}

		while (!file.eof()) {
			if (symb_num != 4) {
				file >> noskipws >> symb;

				if (symb_num == 0)
					v1 = symb;

				else if (symb_num == 2)
					v2 = symb;
			}

			else {
				file >> cap;
				M.at(V.get_index((char)v1)).set(V.get_index((char)v2), cap);
			}

			symb_num++;
			if (symb == '\n' || file.eof())
				symb_num = 0;

		}

		for (int i = 0; i < M.get_size(); i++) {
```

```
                        temp_list.reset();
                        for (int j = 0; j < M.at(i).get_size(); j++) {
                                temp_list.push_back(M.at(i).at(j));
                        }
                        adj.push_back(temp_list);
                }

                for (int i = 0; i < adj.get_size(); i++) {
                        for (int j = i; j < adj.at(i).get_size(); j++) {
                                adj.at(j).set(i, adj.at(i).at(j));
                        }
                }

                temp_list.clear();
        }

        cout << "Max flow: " << maxflow(V, M, adj);

        return 0;
}
```

Class.h

```
#include <stdexcept>
#include <cstdio>

using namespace std;

template <class T>
class Node {
private:
        Node* next;

        T data;

public:
        Node() {
                next = nullptr;
        }
        explicit Node(const T value) {
                next = nullptr;
                data = value;
        }
        ~Node() {
                next = nullptr;
        }
        void set_next(Node* const next_el) {
                next = next_el;
        };
        void set_data(T new_data) {
                data = new_data;
        };
        Node* get_next() {
                return next;
        };
        T get_data() {
                return data;
        };
        void clr_next() {
                delete this;
                next = nullptr;
        };
};

template <class T>
class Queue {
private:
        Node<T>* front;
        Node<T>* back;
```

```
        Node<T>* temp;

        size_t queue_size;

public:
        Queue() {
                front = nullptr;
                back = nullptr;
                temp = nullptr;
                queue_size = 0;
        }

        ~Queue() {
                while (queue_size != 0) {
                        this->takeTop();
                }
                front = nullptr;
                back = nullptr;
                temp = nullptr;
                queue_size = 0;
        }

        Node<T>* get_front() {
                return front;
        }

        Node<T>* get_back() {
                return back;
        }

        void push(T data) {
                temp = new Node<T>(data);
                if (queue_size == 0) {
                        front = temp;
                        back = temp;
                        temp = nullptr;
                        queue_size++;
                        return;
                }
                back->set_next(temp);
                back = back->get_next();

                temp = nullptr;
                queue_size++;
        }

        T takeTop() {
                if (queue_size == 0) {
                        throw std::out_of_range("The queue is empty");
                }
                T front_data;
                front_data = front->get_data();
                if (queue_size == 1) {
                        front = nullptr;
                        back = nullptr;
                        temp = nullptr;
                        queue_size = 0;
                        return front_data;
                }
                temp = front;
                front = front->get_next();
                temp->clr_next();

                temp = nullptr;
                queue_size--;
                return front_data;
        };

        T getTop() {
```

```
		if (queue_size == 0) {
			throw  std::out_of_range("The queue is empty");
		}
		return front->get_data();
	}

	size_t get_size() {
		return queue_size;
	}

};
```

EK.h

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

class Edmonds_Karp
{
public:
	struct pair {
		int first;
		long second;
	};
	List<char> listofvertexes;
	Matrix<int> capacity;
	Matrix<int> adj;
	long INF = 100000;

	~Edmonds_Karp() {
		capacity.clear();
		adj.clear();
		listofvertexes.clear();
	}

	int maxflow() {
		if (listofvertexes.isEmpty())
			throw invalid_argument("Please enter data for the graph");
		int maximum_flow = 0;
		List<int> parent(capacity.get_size(), 0);
		int new_flow;
		if (bfs(0, listofvertexes.get_size() - 1, parent) == 0)
			throw invalid_argument("There are no ways from S to T at all");

		while (new_flow = bfs(0, listofvertexes.get_size() - 1, parent)) {
			maximum_flow += new_flow;
			int cur = listofvertexes.get_size() - 1;
			while (cur != 0) {
				int prev = parent.at(cur);
				capacity.at(prev).set(cur, capacity.at(prev).at(cur) - new_flow);
				capacity.at(cur).set(prev, capacity.at(cur).at(prev) + new_flow);
				cur = prev;
			}
		}
		return maximum_flow;
	}

	void input(string input) {
		ifstream file(input);
		if (!file.is_open())
			throw runtime_error("File was not found, check its name/location");
		int symb_num = 0;
		char symb;
		int cap;
		char first_vrtx, scnd_vrtx;
```

```
		//reading the vertexes list
		while (!file.eof()) {
			file >> noskipws >> symb;
			if ((symb_num == 0 || symb_num == 2) && !listofvertexes.contains(symb))
				listofvertexes.push_back(symb);
			symb_num++;
			if (symb == '\n' || file.eof())
				symb_num = 0;
		}

		if (listofvertexes.at(0) != 'S' || listofvertexes.at(listofvertexes.get_size() -
1) != 'T')
			throw invalid_argument("Please start your graph with S and finish with
T");

		file.clear();
		file.seekg(0);
		List<int> temp;

		for (int i = 0; i < listofvertexes.get_size(); i++) {
			for (int j = 0; j < listofvertexes.get_size(); j++)
				temp.push_back(0);
			capacity.push_back(temp);
			temp.reset();
		}

		//reading the avaliable capacity
		while (!file.eof())
		{
			if (symb_num != 4) {
				file >> noskipws >> symb;

				if (symb_num == 0)
					first_vrtx = symb;

				else if (symb_num == 2)
					scnd_vrtx = symb;
			}

			else {
				file >> cap;

	capacity.at(listofvertexes.get_index((char)first_vrtx)).set(listofvertexes.get_index((c
har)scnd_vrtx), cap);
			}

			symb_num++;
			if (symb == '\n' || file.eof())
				symb_num = 0;

		}

		for (int i = 0; i < capacity.get_size(); i++) {
			temp.reset();
			for (int j = 0; j < capacity.at(i).get_size(); j++) {
				temp.push_back(capacity.at(i).at(j));
			}
			adj.push_back(temp);
		}

		for (int i = 0; i < adj.get_size(); i++) {
			for (int j = i; j < adj.at(i).get_size(); j++) {
				adj.at(j).set(i, adj.at(i).at(j));
			}
		}


		temp.reset();
```

```
		List<int> parent(capacity.get_size(), 0);

	}

	int bfs(int s, int t, List<int>& parent) {
		//s - source, t - sink
		for (int i = 0; i < parent.get_size(); i++)
			parent.set(i, -1);
		parent.set(s, -2);
		Queue <pair> queue;
		queue.push({ s, INF });
		while (queue.get_size() != 0) {
			int cur = queue.getTop().first;
			int flow = queue.getTop().second;
			queue.takeTop();

			for (int next = 0; next < adj.at(cur).get_size(); next++) {
				if (parent.at(next) == -1 && capacity.at(cur).at(next)) {
					parent.set(next, cur);
					int new_flow;
					if (capacity.at(cur).at(next) < flow)
						new_flow = capacity.at(cur).at(next);
					else
						new_flow = flow;
					if (next == t)
						return new_flow;
					queue.push({ next, new_flow });
				}
			}
		}

		//if there are no ways from s to t
		return 0;
	}
};
```

Karp.h

```
#pragma once
#include "Matrix.h"
#include "Class.h"
#include "List.h"
#include <fstream>

long INF = 100000;

int bfs(int s, int t, Matrix<int>& M, Matrix<int>& adj, List<int>& temp_list) {
	for (int i = 0; i < temp_list.get_size(); i++)
		temp_list.set(i, -1);
	temp_list.set(s, -2);

	struct pair {
		long first;
		long second;
	};

	Queue <pair> queue;
	queue.push({ s, INF });
	while (queue.get_size() != 0) {
		int curr = queue.getTop().first;
		int flow = queue.getTop().second;
		queue.takeTop();

		for (int next = 0; next < adj.at(curr).get_size(); next++) {
			if (temp_list.at(next) == -1 && M.at(curr).at(next)) {
				temp_list.set(next, curr);
				int new_flow;
				if (M.at(curr).at(next) < flow)
					new_flow = M.at(curr).at(next);
```

```
                                else
                                        new_flow = flow;
                                if (next == t)
                                        return new_flow;
                                queue.push({ next, new_flow });
                        }
                }
        }
        return 0;
}

int maxflow(List<char>& V, Matrix<int>& M, Matrix<int>& adj) {
        if (V.isEmpty())
                throw invalid_argument("Graph is empty");
        int max_flow = 0;
        List<int> parent(M.get_size(), 0);
        int new_flow;
        if (bfs(0, V.get_size() - 1, M, adj, parent) == 0)
                throw invalid_argument("No path S->T");

        while (new_flow = bfs(0, V.get_size() - 1, M, adj, parent)) {
                max_flow += new_flow;
                int cur = V.get_size() - 1;
                while (cur != 0) {
                        int prev = parent.at(cur);
                        M.at(prev).set(cur, M.at(prev).at(cur) - new_flow);
                        M.at(cur).set(prev, M.at(cur).at(prev) + new_flow);
                        cur = prev;
                }
        }
        return max_flow;
}
```

List.h

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

template<class T>
class List
{
private:
        class Node
        {
        private:
                T data;
                Node* next, * prev;
        public:

                Node() : next(nullptr), prev(nullptr) {};

                Node(T data) {
                        this->data = data;
                        next = nullptr;
                        prev = nullptr;
                }

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

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

                T get_data() {
                        return data;
```

```
		}

		Node* get_next() {
			return next;
		}

		Node* get_prev() {
			return prev;
		}

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

		void set_prev(Node* pointer) {
			prev = pointer;
		}

	};

	Node* head, * tail;

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

public:
	List() : head(nullptr), tail(nullptr) {}

	List(int size, int value) {
		while (size--) {
			push_back(value);
		}
	}

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

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

```
        }
        head = nullptr;
}

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

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

void push_back(List<bool> ls2)
{
        Node* temp = ls2.head;
        size_t n = ls2.get_size();
        while ((temp) && (n--))
        {
                push_back(temp->get_data());
                temp = temp->get_next();
        }
}

void push_front(List& ls2)
{
        Node* temp = ls2.tail;
        size_t n = ls2.get_size();
        while ((temp) && (n--))
        {
                push_front(temp->get_data());
                temp = temp->get_prev();
        }
}

void pop_back()
{
        if (head != tail)
        {
                Node* temp = tail;
                tail = tail->get_prev();
                tail->set_next(nullptr);
                delete temp;
        }
```

```
        else if (!isEmpty())
        {
                Node* temp = tail;
                tail = head = nullptr;
                delete temp;
        }
        else
                throw out_of_range("The list is empty");
}

void pop_front()
{
        if (head != tail)
        {
                Node* temp = head;
                head = head->get_next();
                head->set_prev(nullptr);
                delete temp;
        }
        else if (!isEmpty())
        {
                Node* temp = head;
                head = tail = nullptr;
                delete temp;
        }
        else
                throw out_of_range("The list is empty");
}

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

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

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

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

bool contains(T symb) {
	Node* temp = head;
	while (temp) {
		if (temp->get_data() == symb)
			return true;
		temp = temp->get_next();
	}
	return false;
}

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

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

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

size_t get_index(T symb) {
	Node* temp = head;
	size_t index = 0;
	while (temp) {
		if (temp->get_data() == symb)
			break;
		temp = temp->get_next();
		index++;
	}
```

```
		return index;
	}

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

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

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

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

	void reset() {
		head = nullptr;
		tail = nullptr;
	}

	void reverse() {
		Node* temp1 = head, * temp2 = tail;
		T value;
		for (size_t i = 0; i < get_size() / 2; i++) {
			value = temp1->get_data();
			temp1->set_data(temp2->get_data());
			temp2->set_data(value);
			temp1 = temp1->get_next();
			temp2 = temp2->get_prev();
		}
	}

	int get_price(size_t vertex1, size_t vertex2) {
		Node* temp = head;
		while (temp) {
			if (temp->get_data().first_vertex == vertex1 && temp-
>get_data().scnd_vertex == vertex2)
				return temp->get_data().price;
			temp = temp->get_next();
		}
	}
};
```

Matrix.h

```
#pragma once
```

```cpp
#include "List.h"

//List<List<T>>

template<class T>
class Matrix {
	class Node {
	private:
		List<T> data;
		Node* next, * prev;
	public:

		Node(List<T> data) {
			this->data = data;
			next = nullptr;
			prev = nullptr;
		}
		~Node() {
			next = nullptr;
			prev = nullptr;
		}

		Node() : next(nullptr), prev(nullptr) {};

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

		List<T>& get_data() {
			return data;
		}

		Node* get_next() {
			return next;
		}

		Node* get_prev() {
			return prev;
		}

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

		void set_prev(Node* temp) {
			prev = temp;
		}
	};

	Node* head, * tail;

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

```
		}
	}
public:
	Matrix() : head(nullptr), tail(nullptr) {}

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

	~Matrix()
	{
		while (head)
		{
			tail = head->get_next();
			delete head;
			head = tail;
		}
		head = nullptr;
	}

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

	void push_back(List<T>& data)
	{
		Node* temp = new Node;
		temp->set_data(data);

		if (head)
		{
			temp->set_prev(tail);
			tail->set_next(temp);
			tail = temp;
		}
		else
		{
			head = temp;
			tail = head;
		}
	}

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

	void push_back(Matrix<bool> scnd_list)
```

```
{
	Node* temp = scnd_list.head;
	size_t size = scnd_list.get_size();
	while ((temp) && (size--))
	{
		push_back(temp->get_data());
		temp = temp->get_next();
	}
}

void push_front(Matrix& scnd_list)
{
	Node* temp = scnd_list.tail;
	size_t n = scnd_list.get_size();
	while ((temp) && (n--))
	{
		push_front(temp->get_data());
		temp = temp->get_prev();
	}
}

void insert(size_t index, List<T> data)
{
	Node* temp;
	temp = get_node(index);
	if (temp == head)
		push_front(data);
	else
	{
		Node* newel = new Node;
		newel->set_data(data);
		temp->get_prev()->set_next(newel);
		newel->set_prev(temp->get_prev());
		newel->set_next(temp);
		temp->set_prev(newel);
	}
}

void pop_back()
{
	if (head != tail)
	{
		Node* temp = tail;
		tail = tail->get_prev();
		tail->set_next(nullptr);
		delete temp;
	}

	else if (!isEmpty())
	{
		Node* temp = tail;
		tail = head = nullptr;
		delete temp;
	}
	else
		throw out_of_range("The list is empty");
}

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

```
                Node* temp = head;
                head = tail = nullptr;
                delete temp;
        }
        else
                throw out_of_range("The list is empty");
}

void remove(List<T> data) {
        Node* temp = head;
        while (temp && temp->get_data() != data)
                temp = temp->get_next();
        if (!temp)
                throw out_of_range("Invalid argument");
        if (temp == head)
                pop_front();
        else if (temp == tail)
                pop_back();
        else
        {
                temp->get_prev()->set_next(temp->get_next());
                temp->get_next()->set_prev(temp->get_prev());
                delete temp;
        }
}

List<T>& at(size_t index) {
        Node* temp;
        temp = get_node(index);
        return temp->get_data();
}

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

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

void print() {
        Node* temp = head;
        while (temp) {
                temp->get_data().print();
                temp = temp->get_next();
        }
        std::cout << std::endl;
}

void clear() {
        while (head)
```

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

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

Input.txt

```
S O 3
S P 3
O Q 3
O P 2
P R 2
Q R 4
Q T 2
R T 3
```