МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ ФБГОУ ВО «ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра «Программная инженерия»

**Отчёт**

По лабораторной работе №2

**«Многосвязный список»**

**Выполнили:**

Студенты группы 92-ПГ

Погосян Ж.В.

**Проверили:**

Ужаринский А.Ю.

Орёл — 2021

**Код программы:**

**Main:**

#include <iostream>

#include <queue>

#include <string.h>

#include <string>

#include <vector>

#include "list.h"

using namespace std;

const string UNI\_CITY = "Moscow";

struct abiturient {

string surname;

int marks[3];

int certificate\_with\_honors;

string town;

int dormitory;

bool operator==(abiturient abi) {

return (this->surname.compare(abi.surname) == 0);

}

};

class predicate {

public:

vector<int> operator()(abiturient abi) {

vector<int> criteria;

if (abi.marks[0] == 5 && abi.marks[0] == abi.marks[1] && abi.marks[1] == abi.marks[2])

criteria.push\_back(1);

if (abi.certificate\_with\_honors)

criteria.push\_back(2);

if (abi.town.compare(UNI\_CITY) == 0)

criteria.push\_back(3);

if (abi.dormitory)

criteria.push\_back(4);

return criteria;

}

};

int main() {

List<abiturient, predicate> list;

list.InitList();

setlocale(LC\_ALL, "rus");

int input = 0;

while (true) {

cout << "1 - добавить нового абитуриента" << endl;

cout << "2 - вывести список по критерию" << endl;

cout << "3 - вывести полный список абитуриентов" << endl;

cout << "4 - удалить абитуриента" << endl;

cout << "5 - очистить список" << endl;

cout << "6 - выйти\n";

cin>>input;

switch (input) {

case 1: {

abiturient abi;

cout << "Введите фамилию абитуриента: ";

cin >> abi.surname;

cout << "Первая оценка за вступительный экзамен: ";

cin >> abi.marks[0];

cout << "Вторая оценка за вступительный экзамен: ";

cin >> abi.marks[1];

cout << "Третья оценка за вступительный экзамен: ";

cin >> abi.marks[2];

cout << "Сертификат с отличием(0/1): ";

cin >> abi.certificate\_with\_honors;

cout << "Город, в котором проживает абитуриент: ";

cin >> abi.town;

cout << "Требуется общежитие(0/1): ";

cin >> abi.dormitory;

try {

list.Insert(abi);

cout << "Абитуриент успешно добавлен";

}

catch (bad\_alloc) {

cout << "Невозможно вставить данные. Переполнение физической памяти";

}

break;

}

case 2: {

char input1;

cout << "1 - вывести абитуриентов, которые сдали все экзамены на отлично" << endl;

cout << "2 - вывести абитуриентов, которые имеют аттестат с отличием" << endl;

cout << "3 - вывести абитуриентов, которые проживают в другом городе" << endl;

cout << "4 - вывести абитуриентов, которые нуждаются в общежитии" << endl;

cin >> input1;

if (input1 < '1' || input1 > '4') {

cout << "Неверная опция";

break;

}

int criteria = static\_cast<int>(input1 - 48);

vector<abiturient> result = list.GetByCriteria(criteria);

if (result.empty()) {

cout << "Абитуриентов, удовлетворяющих запросу, не найдено";

input1 = getchar();

}

else {

for (auto abi : result) {

cout << abi.surname << ", проживающий в " << abi.town << endl;

}

}

break;

}

case 3: {

vector<abiturient> result = list.GetAll();

for (auto abi : result)

cout << abi.surname << ", проживающий в городе " << abi.town << endl;

break;

}

case 4: {

abiturient abi;

cout << "Введите фамилию абитуриента: ";

cin >> abi.surname;

if (list.Delete(abi)) {

cout << "Абитуриент успешно удалён";

}

else {

cout << "Абитуриент не найден";

}

break;

}

case 5: {

list.DropList();

cout << "Список успешно сброшен";

break;

}

case 6: {

return 0;

}

}

}

}

**List.h:**

#pragma once

#include <vector>

using namespace std;

template <typename T, typename K>

class List {

public:

void InitList();

void Insert(T data);

bool Delete(T data);

void DropList();

std::vector<T> GetAll();

std::vector<T> GetByCriteria(int criteria);

bool Empty();

~List();

private:

template <typename J>

class Node {

public:

Node() : data(J()) {

for (int i = 0; i < 5; i++)

ptr[i] = nullptr;

}

J data;

Node<J>\* ptr[5];

};

struct descriptor {

Node<T>\* head;

Node<T>\* tail;

Node<T>\* current;

};

bool find(T data);

descriptor descriptor\_array[5];

K predicate;

int size;

};

template <typename T, typename K>

void List<T, K>::InitList() {

size = 0;

for (int i = 0; i < 5; i++) {

descriptor\_array[i].head = nullptr;

descriptor\_array[i].tail = nullptr;

descriptor\_array[i].current = nullptr;

}

}

template <typename T, typename K>

void List<T, K>::Insert(T data) {

vector<int> criteria = predicate(data);

Node<T>\* newNode = new Node<T>;

newNode->data = data;

if (Empty()) {

descriptor\_array[0].head = descriptor\_array[0].tail = newNode;

for (auto x : criteria) {

descriptor\_array[x].head = descriptor\_array[x].tail = newNode;

}

}

else {

descriptor\_array[0].tail->ptr[0] = newNode;

descriptor\_array[0].tail = newNode;

for (auto x : criteria) {

if (descriptor\_array[x].tail == nullptr) {

descriptor\_array[x].head = descriptor\_array[x].tail = newNode;

}

else {

descriptor\_array[x].tail->ptr[x] = newNode;

descriptor\_array[x].tail = newNode;

}

}

}

size++;

}

template <typename T, typename K>

bool List<T, K>::Delete(T data) {

if (Empty() || !find(data))

return false;

Node<T>\* to\_delete;

if (descriptor\_array[0].current->data == data)

to\_delete = descriptor\_array[0].current;

else

to\_delete = descriptor\_array[0].current->ptr[0];

for (int i = 0; i < 5; i++) {

if (descriptor\_array[i].current == nullptr)

break;

if (descriptor\_array[i].current->data == data) {

descriptor\_array[i].head = descriptor\_array[i].current->ptr[i];

if (descriptor\_array[i].head == nullptr)

descriptor\_array[i].tail = nullptr;

}

else {

descriptor\_array[i].current->ptr[i] = descriptor\_array[i].current->ptr[i]->ptr[i];

if (descriptor\_array[i].current->ptr[i] == nullptr)

descriptor\_array[i].tail = descriptor\_array[i].current;

}

descriptor\_array[i].current = nullptr;

}

delete to\_delete;

size--;

return true;

}

template <typename T, typename K>

void List<T, K>::DropList() {

descriptor\_array[0].current = descriptor\_array[0].head;

while (descriptor\_array[0].current) {

Node<T>\* to\_delete = descriptor\_array[0].current;

descriptor\_array[0].current = descriptor\_array[0].current->ptr[0];

delete to\_delete;

}

for (int i = 0; i < 5; i++) {

descriptor\_array[i].head =

descriptor\_array[i].current =

descriptor\_array[i].tail = nullptr;

}

size = 0;

}

template <typename T, typename K>

vector<T> List<T, K>::GetAll() {

vector<T> result;

descriptor\_array[0].current = descriptor\_array[0].head;

while (descriptor\_array[0].current) {

result.push\_back(descriptor\_array[0].current->data);

descriptor\_array[0].current = descriptor\_array[0].current->ptr[0];

}

return result;

}

template <typename T, typename K>

vector<T> List<T, K>::GetByCriteria(int criteria) {

if (criteria == 0 || criteria > 4)

return vector<T>();

vector<T> result;

descriptor\_array[criteria].current = descriptor\_array[criteria].head;

while (descriptor\_array[criteria].current) {

result.push\_back(descriptor\_array[criteria].current->data);

descriptor\_array[criteria].current = descriptor\_array[criteria].current->ptr[criteria];

}

return result;

}

template <typename T, typename K>

bool List<T, K>::find(T data) {

bool found = false;

for (int i = 0; i < 5; i++) {

descriptor\_array[i].current = descriptor\_array[i].head;

while (descriptor\_array[i].current) {

if (descriptor\_array[i].current->data == data || (descriptor\_array[i].current->ptr[i] != nullptr &&

descriptor\_array[i].current->ptr[i]->data == data)) {

found = true;

break;

}

descriptor\_array[i].current = descriptor\_array[i].current->ptr[i];

}

}

return found;

}

template <typename T, typename K>

bool List<T, K>::Empty() {

return (size == 0);

}

template <typename T, typename K>

List<T, K>::~List() {

DropList();

}