**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Списки и их реализация»**

**Вариант 19(3)**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Плюснина Е.Ю. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Постановка задачи**

Реализовать класс односвязного списка с набором методов. (3 вариант)

# Описание реализуемого класса, оценка временной сложности

class List – класс односвязного списка.

Node\* head – указатель на начало списка.

Node\* last – указатель на конец списка.

size\_t size – размер списка.

class Node – элемент списка (вложенный класс)

int data – данные, хранящиеся в списке

Node\* next – указатель на следующий элемент списка.

**Функции:**

void push\_back(int) – добавление в конец списка. Оценка временной сложности: О(1)

void push\_front(int) – добавление в начало списка. Оценка временной сложности: О(1)

void pop\_back() – удаление последнего элемента. Оценка временной сложности: О(1)

void pop\_front() – удаление первого элемента. Оценка временной сложности: О(1)

void insert(int, size\_t); – добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу). Оценка временной сложности: О(n)

int at(size\_t); – получение элемента по индексу. Оценка временной сложности: О(n)

void remove(size\_t); – удаление элемента по индексу. Оценка временной сложности: О(n)

size\_t get\_size() – получение размера списка. Оценка временной сложности: О(1)

void print\_to\_console(); – вывод элементов списка в консоль через разделитель, не использовать at. Оценка временной сложности: О(n)

void clear(); – удаление всех элементов списка. Оценка временной сложности:

void set(int, size\_t) – замена элемента по индексу на передаваемый элемент. Оценка временной сложности: О(n)

bool IsEmpty() – проверка на пустоту списка. Оценка временной сложности: О(1)

void insert(List\*, size\_t); – вставка другого списка в список, начиная с индекса. 14 функция. Оценка временной сложности: О(n2)

# Описание реализованных unit-тестов

TEST\_METHOD(check\_push\_back) – проверка вставки в пустой список

TEST\_METHOD(check\_push\_back\_2) – проверка вставки в НЕ пустой список

TEST\_METHOD(check\_push\_back\_size) – проверка размера

TEST\_METHOD(check\_push\_front) – проверка вставки в пустой список

TEST\_METHOD(check\_push\_front\_2) – проверка вставки в НЕ пустой список

TEST\_METHOD(check\_push\_front\_size) – проверка размера

TEST\_METHOD(check\_pop\_back) – проверка удаления из пустого списка

TEST\_METHOD(check\_pop\_back\_2) – проверка удаления из НЕ пустого списка

TEST\_METHOD(check\_pop\_back\_head) – проверка удаления первого элемента списка (на пустоту списка)

TEST\_METHOD(check\_pop\_back\_size) – проверка размера

TEST\_METHOD(check\_pop\_front) – проверка удаления из пустого списка

TEST\_METHOD(check\_pop\_front\_2) – проверка удаления из НЕ пустого списка

TEST\_METHOD(check\_pop\_front\_head) – проверка удаления первого элемента списка (на пустоту списка)

TEST\_METHOD(check\_pop\_front\_size) – проверка размера

TEST\_METHOD(check\_insert) – проверка вставки в пустой список

TEST\_METHOD(check\_insert\_size) – проверка размера

TEST\_METHOD(check\_at) – проверка работоспособности функции

TEST\_METHOD(check\_at\_2) – проверка выхода за границы списка

TEST\_METHOD(check\_remove) – проверка на удаление списка за границами списка

TEST\_METHOD(check\_remove\_index) – проверка на удаление

TEST\_METHOD(check\_remove\_size) – проверка размера

TEST\_METHOD(check\_get\_size) – проверка размера и работоспособности функции

TEST\_METHOD(check\_clear) – проверка работоспособности функции

TEST\_METHOD(check\_IsEmpty) – проверка на пустоту списка

TEST\_METHOD(chec\_IsNotEmpty) – проверка на НЕ пустоту списка

TEST\_METHOD(check\_insert14) – проверка вставки списка в список перед элементом

# Листинг

**List.h**

#pragma once

#include <iostream>

class List

{

private:

class Node

{

public:

Node(int data = 0, Node\* next = nullptr)

{

this->data = data;

this->next = next;

};

~Node() {};

int data;

Node\* next;

};

Node\* head;

Node\* last;

size\_t size;

public:

List(Node\* head = NULL, Node\* last = NULL)

{

size = 0;

this->head = head;

this->last = last;

}

void push\_back(int); //addition to the end of the list

void push\_front(int); //addition to the top of the list

void pop\_back(); //deleting the last element

void pop\_front(); //deleting of the first element

void insert(int, size\_t); //adding an element by index (insert before the element which was previously available by this index)

int at(size\_t); //retrieve an element by index

void remove(size\_t); //item deletion by index

size\_t get\_size() { //getting the list size

return size;

}

void print\_to\_console(); //output the list elements to the console via delimiter

void clear(); //deleting of all the elements of the list

void set(int, size\_t); //replacement of the element by the index by the passed element

bool IsEmpty(); //check for emptiness of the list

void insert(List\*, size\_t); //insertion of another list in the list, starting from the index (14 function)

~List();

};

**Prog.cpp**

#include "List.h"

#include <iostream>

using namespace std;

void List::push\_back(int new\_elem) //addition to the end of the list

{

if (size != 0) {

last->next = new Node(new\_elem);

last = last->next;

}

else {

head = new Node(new\_elem);

last = head;

}

size++;

}

void List::push\_front(int new\_elem) { //addition to the top of the list

if (size != 0) {

head = new Node(new\_elem, head);

}

else {

head = new Node(new\_elem);

last = head;

}

size++;

}

void List::pop\_back() { //deleting the last element

if (IsEmpty()) {

throw out\_of\_range("You are trying to remove an item from an empty list");

return;

}

else {

if (size == 1) {

delete head;

head = nullptr;

last = nullptr;

}

else {

Node\* cur = head;

while (cur->next != last) {

cur = cur->next;

}

cur->next = nullptr;

delete last;

last = cur;

}

size--;

}

}

void List::pop\_front() { //deleting of the first element

if (IsEmpty()) {

throw out\_of\_range("You are trying to remove an item from an empty list");

return;

}

else {

if (size == 1) {

delete head;

head = nullptr;

last = nullptr;

}

else {

Node\* cur = head;

head = cur->next;

delete cur;

}

size--;

}

}

void List::insert(int insert\_data, size\_t index) { //adding an element by index (insert before the element which was previously available by this index)

if (index == 0) {

push\_front(insert\_data);

}

else if (index >= size) {

throw out\_of\_range("Index is greater than list size");

}

else {

Node\* cur = head;

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

cur = cur->next;

}

Node\* insert\_Node = new Node(insert\_data, cur->next);

cur->next = insert\_Node;

size++;

}

}

int List::at(size\_t n) { //retrieve an element by index

if (n >= size) {

throw out\_of\_range("Index is greater than list size");

}

else {

Node\* cur = head;

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

cur = cur->next;

}

return cur->data;

}

}

void List::remove(size\_t index) { //item deletion by index

if (index >= size) {

throw out\_of\_range("Index is greater than list size");

}

if (index == 0) {

pop\_front();

}

else {

Node\* cur = head;

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

cur = cur->next;

}

Node\* delete\_elem = cur->next;

cur->next = delete\_elem->next;

delete delete\_elem;

size--;

}

}

void List::print\_to\_console() { //output the list elements to the console via delimiter

Node\* cur = head;

while (cur->next != last) {

cout << cur->data << ' ';

cur = cur->next;

}

cout << endl;

}

void List::clear() { //deleting of all the elements of the list

if (size == 0) {

throw out\_of\_range("You are trying to remove an item from an empty list");

}

Node\* cur = new Node;

cur = head;

while (cur != NULL) {

head = cur;

cur = cur->next;

delete head;

size--;

}

delete cur;

head = NULL;

last = head;

}

void List::set(int transmittable\_elem, size\_t index) { //replacement of the element by the index by the passed element

if (index >= size) {

throw out\_of\_range("Index is greater than list size");

}

Node\* cur = head;

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

cur = cur->next;

}

cur->data = transmittable\_elem;

}

bool List::IsEmpty() { //check for emptiness of the list

return (head == nullptr);

}

void List::insert(List\* List2, size\_t index) { //insertion of another list in the list, starting from the index (14 function)

if (index > 0) {

for (int i = (\*List2).get\_size() - 1; i >= 0; i--) {

insert((\*List2).at(i), index - 1);

}

} else

for (int i = (\*List2).get\_size() - 1; i >= 0; i--) {

push\_front((\*List2).at(i));

}

}

List::~List() { //clear

clear();

}

**UnitTest.cpp**

#include "CppUnitTest.h"

#include "../Lab1/List.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest

{

TEST\_CLASS(UnitTest)

{

public:

List\* check;

List\* check2;

TEST\_METHOD\_INITIALIZE(setUp) {

check = new List();

}

TEST\_METHOD\_CLEANUP(cleanUp) {

delete check;

}

TEST\_METHOD(check\_push\_back) {

check->push\_back(1);

Assert::AreEqual(check->get\_size(), (size\_t)1);

}

TEST\_METHOD(check\_push\_back\_2) {

check->push\_back(1);

check->push\_back(2);

Assert::AreEqual(check->at(1), 2);

}

TEST\_METHOD(check\_push\_back\_size) {

check->push\_back(1);

check->push\_back(1);

check->push\_back(1);

Assert::AreEqual(check->get\_size(), (size\_t)3);

}

TEST\_METHOD(check\_push\_front) {

check->push\_front(1);

Assert::AreEqual(check->get\_size(), (size\_t)1);

}

TEST\_METHOD(check\_push\_front\_2) {

check->push\_back(2);

check->push\_front(1);

Assert::AreEqual(check->at(0), 1);

}

TEST\_METHOD(check\_push\_front\_size) {

check->push\_front(1);

check->push\_front(1);

check->push\_front(1);

Assert::AreEqual(check->get\_size(), (size\_t)3);

}

TEST\_METHOD(check\_pop\_back) {

try {

check->pop\_back();

}

catch (std::out\_of\_range e) {

Assert::AreEqual("Error", e.what());

}

}

TEST\_METHOD(check\_pop\_back\_2) {

check->push\_back(1);

check->push\_back(2);

check->push\_back(3);

check->pop\_back();

try {

check->at(2);

}

catch (std::out\_of\_range e) {

Assert::AreEqual("Index is greater than list size", e.what());

}

}

TEST\_METHOD(check\_pop\_back\_head) {

check->push\_back(1);

check->pop\_back(); Assert::IsTrue(check->IsEmpty());

}

TEST\_METHOD(check\_pop\_back\_size) {

check->push\_back(1);

check->push\_back(2);

check->push\_back(3);

check->pop\_back();

}

TEST\_METHOD(check\_pop\_front) {

try {

check->pop\_front();

}

catch (std::out\_of\_range e) {

Assert::AreEqual("Error", e.what());

}

}

TEST\_METHOD(check\_pop\_front\_2) {

check->push\_back(1);

check->push\_back(2);

check->push\_back(3);

check->pop\_front();

Assert::AreEqual(check->at(0), 2);

}

TEST\_METHOD(check\_pop\_front\_head) {

check->push\_back(1);

check->pop\_front();

Assert::IsTrue(check->IsEmpty());

}

TEST\_METHOD(check\_pop\_front\_size) {

check->push\_back(1);

check->push\_back(2);

check->push\_back(3);

check->pop\_front();

Assert::AreEqual(check->get\_size(), (size\_t)2);

}

TEST\_METHOD(check\_insert) {

check->insert(1, 0);

Assert::AreEqual(check->get\_size(), (size\_t)1);

}

TEST\_METHOD(check\_insert\_size) {

check->push\_back(2);

check->push\_back(3);

check->insert(1, 0);

Assert::AreEqual(check->get\_size(), (size\_t)3);

}

TEST\_METHOD(check\_at) {

Assert::AreEqual(check->at(0), 3);

}

TEST\_METHOD(check\_at\_2) {

try {

check->at(200);

}

catch (std::out\_of\_range e) {

Assert::AreEqual("Index is greater than list size", e.what());

}

}

TEST\_METHOD(check\_remove) {

check->push\_back(1);

check->remove(0);

Assert::IsTrue(check->IsEmpty());

}

TEST\_METHOD(check\_remove\_index) {

check->push\_back(1);

check->push\_back(2);

check->remove(0);

Assert::AreEqual(check->at(0), 2);

}

TEST\_METHOD(check\_remove\_size) {

check->push\_back(1);

check->push\_back(2);

check->push\_back(3);

check->remove(1);

Assert::AreEqual(check->get\_size(), (size\_t)2);

}

TEST\_METHOD(check\_get\_size) {

Assert::AreEqual(check->get\_size(), (size\_t)0);

}

TEST\_METHOD(check\_clear) {

check->clear();

Assert::AreEqual(check->get\_size(), (size\_t)0);

}

TEST\_METHOD(check\_IsEmpty) {

Assert::IsTrue(check->IsEmpty());

}

TEST\_METHOD(chec\_IsNotEmpty) {

Assert::IsFalse(check->IsEmpty());

}

TEST\_METHOD(check\_insert14) {

check->insert(check2, 1);

Assert::AreEqual(check->get\_size(), (size\_t)0);

}

};

}

# Вывод

Я научилась реализовывать связные списки через классы и работать с ними. Научилась проверять корректную работу реализованных функций через unittest-ы.