**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»**

## Вариант №7

Студент гр. 9309 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Корягин Е.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2020

## Постановка задачи

Реализовать класс односвязных списков с набором методов. Необходимо реализовать наличие unit-тестов ко всем реализуемым методам. А также реализовать конструктор и деструктор.

## Описание реализуемого класса и метода

push\_back - Добавляет элемент в конец списка

push\_front - Добавляет элемент в начало списка

pop\_back - Удаляет последний элемент списка

pop\_front - Удаляет первый элемент списка

insert - Вставляет элемент с указанными данными перед элементом по указателю

at - Возвращает данные из элемента по индексу

remove - Удаляет элемент по индексу

get\_size - Возвращает размер списка

print\_to\_console - Выводит все элементы списка в консоль

clear - Удаляет весь список

set - Изменяет данные элемента по указанному индексу

is\_empty - Проверяет, пуст ли список

push\_front\_l - Вставляет список в начала другого списка

## Оценка временной сложности методов

push\_back = O(N).

push\_front = O(1).

pop\_back = O(N).

pop\_front = O(1).

insert = O(N).

at = O(N).

remove = O(N).

get\_size = O(1).

print\_to\_console = O(N).

clear = O(N).

set = O(N).

isEmpty = O(1).

Push\_front\_l = O(N).

## Описание реализованных unit-тестов

correct\_push\_back – идёт проверка функции push\_back(), создаётся список, после чего размер получившегося списка сравнивается с ожидаемым размером списка.

correct\_push\_front – идёт проверка функции push\_front(), создаётся список, после чего размер получившегося списка сравнивается с ожидаемым размером списка.

correct\_pop\_back – идёт проверка функции pop\_back(), создаётся список, после чего удаляется последний элемент из списка (методом pop\_back()), затем размер получившегося списка сравнивается с ожидаемым размером списка.

correct\_pop\_front – идёт проверка функции pop\_front(), создаётся список, после чего удаляется самый первый элемент из списка (методом pop\_front()), затем размер получившегося списка сравнивается с ожидаемым размером списка.

correct\_insert – идёт проверка функции insert(), создаётся список, после чего вставляется элемент в список перед нужным индексом (методом insert()), затем размер получившегося списка сравнивается с ожидаемым размером списка.

correct\_index\_at – идёт проверка функции at(), создаётся список, после чего методом at() находим элемент, сравниваем его с ожидаемым элементом.

correct\_remove – идёт проверка функции remove(), создаётся список, после чего методом remove() удаляется элемент из списка, затем размер получившегося списка сравнивается с ожидаемым размером списка.

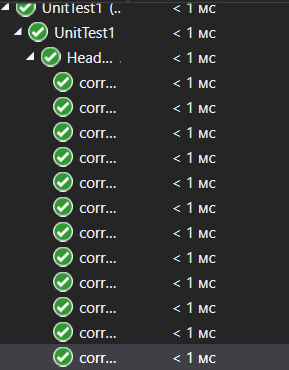
correct\_get\_size – идёт проверка функции get\_size(), создаётся список, после чего методом get\_size() находится размер списка, он сравнивается с ожидаемым размером списка.

correct\_clear – идёт проверка функции clear(), создаётся список, после чего методом clear() очищается весь список, затем размер списка сравнивается с ожидаемым размером списка.

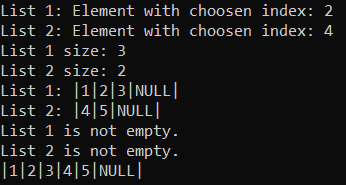
correct\_set – идёт проверка функции set(), создаётся список, после чего методом set() вставляется элемент, затем размер получившегося списка сравнивается с ожидаемым размером списка.

correct\_isEmpty – идёт проверка функции isEmpty(), создаётся список, после чего методом isEmpty() проверяется пустой ли список или нет(переменная bool), затем полечившиеся переменная сравнивается с ожидаемой переменной.

correct\_push\_front\_l – идёт проверка функции push\_front\_l(List), создаются списки, после чего методом push\_front\_l(List) первый список вставляется в начало второго, и итоговый список сравнивается с ранее обозначенным размером.



## Пример работы



## Листинг

**main.cpp**

#include "class\_n\_func.h"

#include <iostream>

#include <stdexcept>

using namespace std;

void main()

{

List\* list1 = new List;

List\* list2 = new List;

list1->push\_back(1);

list1->push\_back(2);

list1->push\_back(3);

list2->push\_back(4);

list2->push\_back(5);

cout << "List 1: Element with choosen index: " << list1->at(1);

cout << endl;

cout << "List 2: Element with choosen index: " << list2->at(0);

cout << endl;

cout << "List 1 size: " << list1->get\_size();

cout << endl;

cout << "List 2 size: " << list2->get\_size();

cout << endl;

cout << "List 1: ";

list1->print\_to\_console();

cout << "List 2: ";

list2->print\_to\_console();

if (list1->isEmpty() == true)

{

cout << "List 1 is not empty.";

cout << endl;

}

else

{

cout << "List 1 is empty.";

cout << endl;

}

if (list2->isEmpty() == true)

{

cout << "List 2 is not empty.";

cout << endl;

}

else

{

cout << "List 2 is empty.";

cout << endl;

}

list1->push\_front\_l(\*list2);

list1->print\_to\_console();

list2->clear();

list1->clear();

delete list1;

list2->clear();

delete list2;

exit(0);

}

**class\_n\_func.h**

pragma once

#include <iostream>

#include <stdexcept>

using namespace std;

class Node

{

int nmbr;

class Node\* next;

friend class List;

};

class List

{

Node\* head;

Node\* last;

int count = 0;

public:

List()

{

head = NULL;

last = head;

}

~List()

{

head = NULL;

last = NULL;

delete head;

delete last;

}

void push\_front(int num)

{

Node\* elem = new Node();

elem->nmbr = num;

count++;

if (head == NULL)

{

elem->next = NULL;

head = elem;

last = head;

}

else

{

elem->next = head;

head = elem;

}

}

void insert(int index, int num)

{

if (index < 0)

{

throw underflow\_error("Index is lower than a minimum size!");

}

if (index > count)

{

throw underflow\_error("Index is higher than a maximum size!");

}

Node\* elem = new Node();

Node\* tmpr;

elem->nmbr = num;

count++;

tmpr = head;

if (index == 0)

{

if (head == NULL)

{

elem->next = NULL;

head = elem;

last = head;

}

else

{

elem->next = head;

head = elem;

}

}

else

{

for (int i = 0; i < index - 1; i++)

{

tmpr = tmpr->next;

}

elem->next = tmpr->next;

tmpr->next = elem;

}

tmpr = NULL;

delete tmpr;

}

void push\_back(int num)

{

Node\* elem = new Node();

Node\* tmpr;

elem->nmbr = num;

count++;

if (head == NULL)

{

elem->next = NULL;

head = elem;

last = head;

}

else

{

tmpr = head;

while (tmpr->next != NULL)

{

tmpr = tmpr->next;

}

elem->next = NULL;

tmpr->next = elem;

last = tmpr;

}

tmpr = NULL;

delete tmpr;

}

void print\_to\_console()

{

Node\* tmpr;

tmpr = head;

cout << "|";

while (tmpr != NULL)

{

cout << tmpr->nmbr << "|";

tmpr = tmpr->next;

}

if (tmpr == NULL)

{

cout << "NULL|";

}

cout << endl;

tmpr = NULL;

delete tmpr;

}

void pop\_back()

{

if (count == 0)

throw underflow\_error("List was already empty!");

last = head;

Node\* tmpr;

tmpr = head;

while (last->next != NULL)

{

last = last->next;

}

if (tmpr != last)

{

while (tmpr->next != last)

{

tmpr = tmpr->next;

}

last->next = NULL;

delete last;

tmpr->next = NULL;

last = tmpr;

}

else

{

delete tmpr;

}

count--;

tmpr = NULL;

delete tmpr;

}

void remove(int index)

{

Node\* tmpr;

if (count > 0)

{

tmpr = head;

if (index == 0)

{

tmpr = head;

head = head->next;

tmpr->next = NULL;

delete tmpr;

}

else if (index == count - 1)

{

if (count == 0)

return;

last = head;

tmpr = head;

while (last->next != NULL)

{

last = last->next;

}

if (tmpr != last)

{

while (tmpr->next != last)

{

tmpr = tmpr->next;

}

last->next = NULL;

delete last;

tmpr->next = NULL;

last = tmpr;

}

else

{

delete tmpr;

}

}

else

{

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

{

tmpr = tmpr->next;

}

last = tmpr->next;

tmpr->next = last;

delete last;

}

count--;

}

tmpr = NULL;

delete tmpr;

}

int at(int index)

{

if (index > count || count < 0 || index < 0)

throw underflow\_error("Index size or list size are lower or higher than acceptable values!");

last = head;

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

{

last = last->next;

}

return last->nmbr;

}

void set(int index, int value)

{

if (index > count || count < 0 || index < 0)

throw underflow\_error("Index size or list size are lower or higher than acceptable values!");

Node\* tmpr;

tmpr = head;

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

{

tmpr = tmpr->next;

}

Node\* elem = new Node();

elem->nmbr = value;

tmpr->nmbr = elem->nmbr;

elem->next = NULL;

delete elem;

tmpr = NULL;

delete tmpr;

}

void pop\_front()

{

if (count == 0)

throw underflow\_error("List was already empty!");

Node\* tmpr;

tmpr = head;

head = head->next;

tmpr->next = NULL;

delete tmpr;

count--;

}

size\_t get\_size()

{

return count;

}

bool isEmpty()

{

return (count != 0);

}

void clear()

{

Node\* tmpr;

for (int j = 0; j < count; j++)

{

tmpr = head;

head = head->next;

tmpr->next = NULL;

delete tmpr;

}

count = 0;

tmpr = NULL;

delete tmpr;

}

void push\_front\_l(List l1)

{

Node\* tmpr;

int temp = l1.count;

count = l1.count + count;

if (head != NULL)

{

last = head;

while (last->next != NULL)

{

last = last->next;

}

}

l1.last = l1.head;

while (temp != 0)

{

Node\* elem = new Node();

elem->nmbr = 0;

if (head == NULL)

{

elem->next = NULL;

head = elem;

last = head;

last->nmbr = l1.last->nmbr;

l1.last = l1.last->next;

}

else

{

tmpr = head;

while (tmpr->next != NULL)

{

tmpr = tmpr->next;

}

elem->next = NULL;

tmpr->next = elem;

last = elem;

last->nmbr = l1.last->nmbr;

l1.last = l1.last->next;

}

temp--;

}

tmpr = NULL;

delete tmpr;

}

};

**UnitTest.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include <stdexcept>

#include "../Lab1\_Koryagin/class\_n\_func.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1

{

TEST\_CLASS(HeaderTests)

{

public:

List\* lst;

List\* lst2;

TEST\_METHOD\_INITIALIZE(setUp)

{

lst = new List();

lst2 = new List();

}

TEST\_METHOD\_CLEANUP(cleanUP)

{

delete lst;

delete lst2;

}

TEST\_METHOD(correct\_push\_back)

{

lst->push\_back(1);

lst2->push\_back(2);

lst->push\_back(3);

lst2->push\_back(4);

lst->push\_back(5);

lst2->push\_back(6);

Assert::AreEqual(lst->get\_size(), lst2->get\_size());

}

TEST\_METHOD(correct\_push\_front)

{

lst->push\_front(3);

lst2->push\_front(1);

lst->push\_front(1);

lst2->push\_front(2);

lst->push\_front(3);

lst2->push\_front(4);

lst->push\_front(5);

lst2->push\_front(6);

Assert::AreEqual(lst->get\_size(), lst2->get\_size());

}

TEST\_METHOD(correct\_pop\_back)

{

lst->push\_back(1);

lst->push\_back(2);

lst->push\_back(3);

lst2->push\_back(1);

lst2->push\_back(2);

lst->pop\_back();

Assert::AreEqual(lst->get\_size(), lst2->get\_size());

}

TEST\_METHOD(correct\_pop\_front)

{

lst->push\_back(3);

lst->push\_back(1);

lst->push\_back(2);

lst2->push\_back(1);

lst2->push\_back(2);

lst->pop\_front();

Assert::AreEqual(lst->get\_size(), lst2->get\_size());

}

TEST\_METHOD(correct\_insert)

{

size\_t size = 3;

lst->push\_front(1);

lst->push\_front(3);

lst->insert(1, 2);

Assert::AreEqual(lst->get\_size(), size);

}

TEST\_METHOD(correct\_index\_at)

{

lst->push\_back(1);

lst->push\_back(2);

lst->push\_back(3);

lst->push\_back(4);

lst->push\_back(5);

lst->push\_back(6);

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

}

TEST\_METHOD(correct\_remove)

{

size\_t size = 4;

lst->push\_front(1);

lst->push\_front(2);

lst->push\_back(3);

lst->push\_back(4);

lst->push\_back(5);

lst->remove(1);

Assert::AreEqual(lst->get\_size(), size);

}

TEST\_METHOD(correct\_get\_size)

{

lst->push\_back(1);

lst->push\_back(2);

lst->push\_back(3);

size\_t size = 3;

Assert::AreEqual(lst->get\_size(), size);

}

TEST\_METHOD(correct\_clear)

{

size\_t size = 0;

lst->push\_front(1);

lst->push\_front(2);

lst->push\_front(3);

lst->clear();

Assert::AreEqual(lst->get\_size(), size);

}

TEST\_METHOD(correct\_set)

{

size\_t size = 3;

lst->push\_front(1);

lst->push\_front(2);

lst->push\_front(3);

lst->set(1, 3);

Assert::AreEqual(lst->get\_size(), size);

}

TEST\_METHOD(correct\_isEmpty)

{

lst->push\_front(1);

lst->push\_front(2);

lst->push\_front(3);

Assert::AreEqual(lst->isEmpty(), true);

}

TEST\_METHOD(correct\_push\_front\_l1)

{

size\_t size = 5;

lst->push\_back(1);

lst->push\_back(2);

lst2->push\_back(3);

lst2->push\_back(4);

lst2->push\_back(5);

lst->push\_front\_l(\*lst2);

Assert::AreEqual(lst->get\_size(), size);

}

};

}