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: Изучение понятие класса и объекта класса C++

Вариант 1

Студент 0301 _____
Сморжок В. Е.

Преподаватель _____
Тутуева А.В.

Санкт-Петербург,

2021

Задание лабораторной работы:

Реализовать класс связного списка с набором методов. Данные, хранящиеся в списке могут быть любого типа на ваш выбор (например, int).

Обязательна реализация конструктора и деструктора.

Список методов, которые реализует каждый вариант (приведено для типа данных int):

1. void push_back(int); // добавление в конец списка
2. void push_front(int); // добавление в начало списка
3. void pop_back(); // удаление последнего элемента
4. void pop_front(); // удаление первого элемента
5. void insert(int, size_t) // добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу)
6. int at(size_t); // получение элемента по индексу
7. void remove(size_t); // удаление элемента по индексу
8. size_t get_size(); // получение размера списка
9. void clear(); // удаление всех элементов списка
10. void set(size_t, int); // замена элемента по индексу на передаваемый элемент
11. bool isEmpty(); // проверка на пустоту списка
12. Перегрузка оператора вывода <<

Список методов, которые реализуются в отдельных вариантах:

13. void reverse(); // меняет порядок элементов в списке на обратный

Текст программы:

```
#include <iostream>

using namespace std;

class List
{
public:

	List(int element) { //creating a constructor with variables
		Node* buffer = new Node; // Create temporary node
		buffer->element = element;
		buffer->next = tail;
		head = buffer;
		tail = buffer;
	}

	List() // default constructor
	{
		Node* buffer = new Node; // Create temporary node
		buffer->next = NULL;
		buffer->element = NULL;
		head = tail = NULL;
	}
	~List() { // destructor
		clear();
	}

	void Print() // additional function for output
	{
		Node* buffer = head;
		cout << "List: ";
		while (buffer != NULL)
		{
			cout << buffer->element << " ";
			buffer = buffer->next;
		}
		cout << endl;
	}

	void push_back(int element) // addition in the end of list
	{
		Node* buffer = head;
		if (buffer->element == NULL)
		{
			buffer->element = element;
			head = buffer;
		}
		else
		{
			while (buffer->next != NULL)
			{
				buffer = buffer->next;
			}
			Node* penultimate = buffer;
			Node* buffer1 = new Node;
			penultimate->next = buffer1;
			buffer1->next = NULL;
			buffer1->element = element;
```

```
                tail = buffer1;
        }
}
void push_front(int element) // addition in the beggining of list
{
        Node* buffer = new Node;
        int index = get_size();
        if (index != 0)
        {
                buffer->next = head;
                head = buffer;
                buffer->element = element;
        }
        else
        {
                buffer->element = element;
                buffer->next = NULL;
                head = tail = buffer;
        }
}
void pop_back() // delete last element
{
        Node* buffer = head;
        int index = get_size();
        Node* penultimate = NULL;
        if (index == 0)
        {
                throw invalid_argument("List is empty. Deletion is not possible");
        }
        else {

                if (index == 1)
                {
                        delete(buffer);
                        head = tail = nullptr;
                }
                else {
                        while (buffer->next != NULL)
                        {
                                penultimate = buffer;
                                buffer = buffer->next;
                        }
                        buffer->element = NULL;
                        delete(buffer);
                        if (penultimate != NULL)
                        {
                                penultimate->next = NULL;
                                tail = penultimate;
                        }
                }
        }
}

void pop_front() // delete first element
{
        int index = get_size();
        if (index != 0)
        {
                Node* buffer = head;
                if (index == 1)
                {
```

```
                        delete (buffer);
                        head = tail = nullptr;
                }
                else
                {
                        head = buffer->next;
                        delete(buffer);
                }
        }
        else
        {
                throw invalid_argument("List is empty. Deletion is not possible");
        }
}


void insert(int number, int element) // insert element with index
{
        int index = get_size();
        int check = index;
        if (number <= index && number > 0)
        {
                Node* buffer = head;
                index = 1;
                Node* penultimate = head;
                while (index != number)
                {
                        penultimate = buffer;
                        buffer = buffer->next;
                        index++;
                }
                Node* buffer1 = new Node;
                if (index == 1)
                {
                        buffer1->element = element;
                        buffer1->next = buffer;
                        head = buffer1;
                }
                else
                {
                        buffer1->element = element;
                        buffer1->next = buffer;
                        penultimate->next = buffer1;
                }
        }
        else
        {

                //throw out_of_range("incorrect index");
        }
}

int at(int number) // getting an element by index
{
        Node* buffer = head;
        Node* buffer1 = head;
        int check;
        int index = get_size();
        if (number > index || number <= 0)
        {
                throw out_of_range("Incorrect index");
```

```
	}
	else
	{
		index = 0;
		while (index != number)
		{
			buffer1 = buffer;
			buffer = buffer->next;
			index++;
		}
		check = buffer1->element;
	}
	return check;
}
void remove(int number) // deleting an element by index
{
	int index = get_size();
	if (number <= index && number > 0)
	{
		if (number == 1)
		{
			pop_front();
		}
		else
		{
			if (number == index) pop_back();
			else
			{
				Node* buffer = head;
				Node* penultimate = NULL;
				index = 1;
				while (index != number)
				{
					penultimate = buffer;
					buffer = buffer->next;
					index++;
				}
				penultimate->next = buffer->next;
				delete (buffer);
			}
		}
	}
	else
	{
		throw out_of_range("Incorrect index");
	}
}

size_t get_size() // getting the size of list
{
	Node* buffer = head;
	int index = 0;
	while (buffer != NULL)
	{
		index++;
		buffer = buffer->next;
	}
	if (head == NULL)
	{
		index = 0;
	}
```

```
		return index;
	}
	void clear()// delete all elements of the list
	{
		Node* buffer = head;
		Node* penultimate = head;
		while (buffer != tail)
		{
			penultimate = buffer->next;
			delete (buffer);
			buffer = penultimate;
		}
		head = tail = NULL;
	}
	void set(int element, int number) // replacing the list element by index with the passed
element
	{
		int index = get_size();
		if (number > index || number <= 0)
		{
			throw out_of_range("Incorrect index");
		}
		else
		{
			Node* buffer = head;
			index = 1;
			while (index != number)
			{
				buffer = buffer->next;
				index++;
			}
			buffer->element = element;
		}
	}
	bool isEmpty()// test on empty list
	{
		bool answer = false;
		int index = get_size();
		if (index == 0)
		{
			answer = true;
			cout << "List is empty" << endl;
		}
		else cout << "List isn't empty" << endl;
		return answer;
	}


	void reverse() // the function reverses the order of the elements in the list
	{
		Node* buffer = head;
		Node* penultimate = tail;
		int buffernew; int count = 1;
		int index = get_size();
		for (int i = 1; i <= index; i++)
		{
			buffernew = buffer->element;
			penultimate = head;
			while (count != index)
			{
				penultimate = penultimate->next;
```

```
                        count++;
                }
                buffer->element = penultimate->element;
                penultimate->element = buffernew;
                index--;
                buffer = buffer->next;
                count = 1;
            }
        }

        friend ostream& operator << (ostream& stream, const List& list); // operator overloading



private:
        class Node { // creating new class
        public:
                int element; // value of element
                Node* next; // pointers to next element and previous element
        };
        Node* tail; Node* head;

};
ostream& operator << (ostream& stream, const List& list) // for output
{
        List::Node* print = list.head;
        while (print != NULL) {
                stream << print->element << " ";
                print = print->next;
        }
        cout << endl;
        return stream;
}

int main()
{
        setlocale(LC_ALL, "Russian");
        cout << "Creating a list" << endl;
        List variable(8);
        cout << variable << endl;
        cout << "fill in the list. add 5" << endl;
        variable.push_back(5);
        cout << variable;
        cout << "getting an element by index" << endl;
        try
        {
                cout<< variable.at(1);
        }
        catch (const out_of_range error)
        {
                cout << error.what();
        }
        cout << "Add 2 elements in list" << endl;
        variable.push_front(6);
        variable.push_front(1);
        cout << "List: " << variable << endl;
        cout << "check this operation again" << endl;
        try
        {
                cout << variable.at(1);
        }
```

```
	catch (const out_of_range error)
	{
		cout << error.what();
	}
	cout << "Size of list is: " << endl;
	cout << variable.get_size()<< endl;

	variable.pop_back();
	cout << "delete last element" << endl << "List: " << variable << endl;
	variable.pop_front();
	cout << "delete first element" << endl << "List: " << variable << endl;
	variable.insert(2, 9);
	cout << "insert the list element by index with the passed element" << endl << "List: "
<< variable << endl;
	cout << "replacing the list element by index with the passed element" << endl;
	variable.set(5, 2);
	cout << "List: " << variable << endl;
	variable.push_back(7);
	cout << "ad an element in the end of list" << endl << "List: " << variable << endl;
	variable.reverse();
	cout << "reverse the list" << endl << "List: " << variable << endl;

	variable.remove(1);
	cout << "deleting an element by index" << endl << "List: " << variable << endl;
	cout << "clear the list" << endl;
	variable.clear();
	variable.isEmpty();

	return 0;
}
```

Unit-тест:

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../АиСД_1лаба/List.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest
{
	TEST_CLASS(UnitTest)
	{
	public:
		TEST_METHOD(DefaultConstructorTest)
		{
			List List;
			Assert::IsTrue(List.get_size() == 0);
		}
		TEST_METHOD(ParametrConstructorTest)
		{
			List List(5);
			List::Node* buffer = List.head;
			Assert::IsTrue(buffer->element == 5);
			Assert::IsTrue(List.get_size() == 1);
		}
		TEST_METHOD(TestPush_back)
		{
			List List(5);
			Assert::IsTrue(List.get_size() == 1);
			List.push_back(6);
```

```
		List::Node* buffer = List.head;
		buffer = buffer->next;
		Assert::IsTrue(buffer->element = 6);
	}
	TEST_METHOD(TestPush_front)
	{
		List List(5);
		List.push_back(8);
		Assert::IsTrue(List.get_size() == 2);
		List::Node* buffer = List.head;
		List::Node* buffer1 = List.head;
		buffer1->element = buffer->element;
		List.push_front(9);
		buffer = List.head;
		Assert::IsTrue(buffer->element != buffer1->element);
	}
	TEST_METHOD(Testpop_back)
	{
		List List(5);
		List.push_back(4);
		List.pop_back();
		Assert::IsTrue(List.tail->element == 5);
		Assert::IsTrue(List.get_size() == 1);
	}
	TEST_METHOD(Testpop_front)
	{
		List List(5);
		List.push_back(8);
		Assert::IsTrue(List.get_size() == 2);
		List.pop_front();
		Assert::IsTrue(List.head->element == 8);
		Assert::IsTrue(List.get_size() == 1);
	}
	TEST_METHOD(Testinsert)
	{
		List List(5);
		List.push_back(8);
		List::Node* buffer = List.head;
		List.insert(1, 6);
		Assert::IsTrue(List.head->element == 6);
		List.insert(2, 7);
		buffer = List.head;
		buffer = buffer->next;
		Assert::IsTrue(buffer->element == 7);
		List.insert(3, 2);
		buffer = List.head;
		buffer = buffer->next;
		buffer = buffer->next;
		Assert::IsTrue(buffer->element == 2);
	}
	TEST_METHOD(Testat)
	{
		List List(5);
		List.push_back(8);
		List.push_back(7);
		List::Node* buffer = List.head;
		int check = List.at(2);
		buffer = List.head;
		buffer = buffer->next;
		Assert::IsTrue(buffer->element == check);
		check = List.at(1);
```

```
		Assert::IsTrue(List.head->element == check);
		check = List.at(3);
		Assert::IsTrue(List.tail->element == check);

	}
	TEST_METHOD(Testremove)
	{
		List List(5);
		List.push_back(8);
		List.push_back(4);
		List::Node* buffer = List.head;
		List.remove(1);
		Assert::IsTrue(List.head->element == 8);
		List.push_back(3);
		List.remove(3);
		Assert::IsTrue(List.tail->element == 4);
		List.push_back(0);
		List.remove(2);
		buffer = List.head;
		buffer = buffer->next;
		Assert::IsTrue(buffer->element == 0);
	}

	TEST_METHOD(Testget_size)
	{
		List List(5);
		List.push_back(8);
		Assert::IsTrue(List.get_size() == 2);
	}
	TEST_METHOD(Testclear)
	{
		List List(5);
		List.push_back(8);
		List.clear();
		Assert::IsTrue(List.head == NULL && List.tail == NULL);
	}
	TEST_METHOD(TestisEmpty)
	{
		List List;
		List.isEmpty();
		Assert::IsTrue(List.isEmpty() == true);
		Assert::IsTrue(List.get_size() == 0);
	}
	TEST_METHOD(Testset)
	{
		List List(5);
		List.push_back(9);
		List.push_back(8);
		int check = 6;
		List.set(6, 1);
		List::Node* buffer = List.head;
		Assert::IsTrue(List.head->element == check);
		List.set(5, 3);
		Assert::IsTrue(List.tail->element == 5);
		List.set(2, 2);
		buffer = buffer->next;
		Assert::IsTrue(buffer->element == 2);
	}
	TEST_METHOD(Testreverse)
	{
		List List(5);
```

```
                List.push_back(8);
                List.push_back(3);
                List::Node* buffer = List.head;
                List.reverse();
                Assert::IsTrue(buffer->element == 3);
                buffer = buffer->next;
                Assert::IsTrue(buffer->element == 8);
                buffer = buffer->next;
                Assert::IsTrue(buffer->element == 5);
            }
        };
}
```

Описание реализуемого класса и каждого метода:

1. В разделе класса private выделяется память под каждую составляющую списка и создается узел
2. Конструктор по умолчанию List() заполняет элементы списка, там же определяется положение head и tail, определяются указатели на следующий элемент
3. В конструкторе с переданной константой List(element) заполняются некоторые ячейки элементов списка
4. Метод push_back отвечает за добавление элемента в конце списка. Им можно оперировать для заполнения даже пустого списка
5. Метод push_front отвечает за добавление элемента в начало списка. Перекидывается указатель на головной элемент
6. Метод pop_back отвечает за удаление последнего элемента списка, перекидывается хвост
7. Метод pop_front отвечает за удаление первого элемента списка, перекидывается голова
8. Метод insert отвечает за вставку элемента по индексу, сдвигая следующие элементы. Меняется указатель предыдущего элемента на вставленный, указатель которого указывает на следующий
9. Метод at отвечает за вывод на экран элемента по запрошенному индексу
10. Метод remove отвечает за удаление элемента по индексу. Указатели меняются. Случаи с удалением первого и последнего элемента учтены
11. Функция get_size отвечает за вывод количества элементов списка на экран
12. Метод clear отвечает за очистку списка. Удаление каждого элемента списка

13. Метод set отвечает за замену элемента списка по индексу на передаваемый элемент
14. Метод isEmpty проверяет список на пустоту
15. Метод reverse отвечает за переворачивание всего списка. Указатели меняются. Список становится «перевернутым»

Оценка временной сложности каждого метода

1. Конструктор по умолчанию O(1)
2. Конструктор с константой O(1)
3. Метод push_back O(n)
4. Метод push_front O(1)
5. Метод pop_back O(n)
6. Метод pop_front O(1)
7. Метод insert O(n)
8. Метод at O(n)
9. Метод remove O(n)
10. Метод get_size O(n)
11. Метод clear O(n)
12. Метод set O(n)
13. Метод isEmpty O(1)
14. Метод reverse O(n^2)

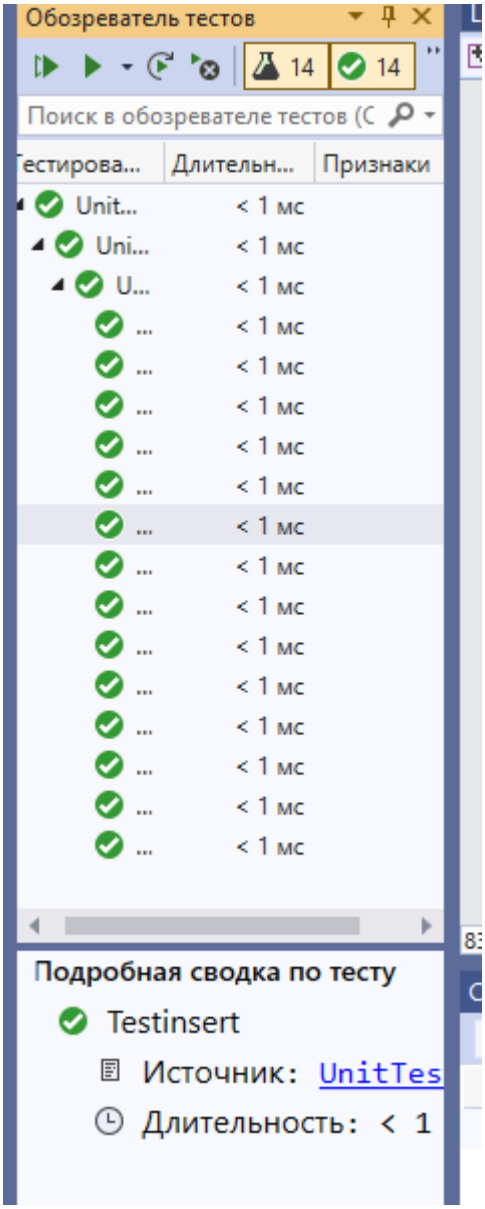
Описание реализованных unit-тестов

1. DefaultConstructorTest проверяет список на количество элементов (= 0)
2. ParametrConstructorTest проверяет список на количество элементов (=1). Проверяет, передано ли первому элементу списка нужное значение
3. TestPush_back проверяет, передано ли следующему элементу списка нужное значение и проверяет на количество элементов (=2)
4. TestPush_front проверяет неравенство первого элемента изначального списка на добавленное. Проверяет на увеличение количества элементов списка

5. Testpop_back проверяет равенство последнего элемента предпоследнему. Проверяет количество элементов списка
6. Testpop_front проверяет равенство первого элемента переданному значению. Проверяет количество элементов списка
7. Testinsert проверяет равенство выбранного элемента по индексу заданному значению. Проверякт количество элементов списка
8. Testat проверяет выведенное на экран число, равное запрошенному элементу списка
9. Testremove проверяет равенство элемента под индексом из изначального списка элементу из измененного списка
10. Testget_size проверяет количество элементов списка
11. Testclear проверяет последний и первый элемент на равенство нулю
12. TestisEmpty проверяет на пустоту списка и проверяет количество элементов (=0)
13. Testset проверяет равенство элемента под индексом переданному значению
14. Testreverse проверяет равенство конца списка началу, середину середине, начало концу

Пример работы

```
Creating a list
fill in the list. add 5
getting an element by index
Add 2 elements in list
List: 1 6 8 5

check this operation again
Size of list is:
4delete last element
List: 1 6 8

List isn't empty
delete first element
List: 6 8

insert the list element by index with the passed element
List: 6 9 8

replacing the list element by index with the passed element
List: 6 5 8

ad an element in the end of list
List: 6 5 8 7

reverse the list
List: 7 8 5 6

List isn't empty
deleting an element by index
List: 8 5 6

clear the list
List is empty
```


Обозреватель тестов
14
14
Поиск в обозревателе тестов (C
Длительн...
Признаки
Unit...
< 1 мс
Uni...
< 1 мс
U...
< 1 мс
Подробная сводка по тесту
Testinsert
Источник: UnitTes
Длительность: < 1

```
TEST_METHOD(DefaultConstructorTest)
{
    List List;
    Assert::IsTrue(List.get_size() == 0);
}
TEST_METHOD(ParametrConstructorTest)
{
    List List(5);
    List::Node* buffer = List.head;
    Assert::IsTrue(buffer->element == 5);
    Assert::IsTrue(List.get_size() == 1);
}
TEST_METHOD(TestPush_back)
{
    List List(5);
    Assert::IsTrue(List.get_size() == 1);
    List.push_back(6);
    List::Node* buffer = List.head;
    buffer = buffer->next;
    Assert::IsTrue(buffer->element = 6);
}
TEST_METHOD(TestPush_front)
{
    List List(5);
    List.push_back(8);
    Assert::IsTrue(List.get_size() == 2);
    List::Node* buffer = List.head;
    List::Node* buffer1 = List.head;
    buffer1->element = buffer->element;
    List.push_front(9);
    buffer = List.head;
    Assert::IsTrue(buffer->element != buffer1->element);
}
TEST_METHOD(Testpop_back)
{
    List List(5);
    List.push_back(4);
    List.pop_back();
    Assert::IsTrue(List.tail->element == 5);
```

```
}
TEST_METHOD(Testpop_front)
{
    List List(5);
    List.push_back(8);
    Assert::IsTrue(List.get_size() == 2);
    List.pop_front();
    Assert::IsTrue(List.head->element == 8);
    Assert::IsTrue(List.get_size() == 1);
}
TEST_METHOD(Testinsert)
{
    List List(5);
    List.push_back(8);
    List::Node* buffer = List.head;
    List.insert(1, 6);
    Assert::IsTrue(List.head->element == 6);
    List.insert(2, 7);
    buffer = List.head;
    buffer = buffer->next;
    Assert::IsTrue(buffer->element == 7);
    List.insert(3, 2);
    buffer = List.head;
    buffer = buffer->next;
    buffer = buffer->next;
    Assert::IsTrue(buffer->element == 2);
}
TEST_METHOD(Testat)
{
    List List(5);
    List.push_back(8);
```

```
TEST_METHOD(Testremove)
{
    List List(5);
    List.push_back(8);
    List.push_back(4);
    List::Node* buffer = List.head;
    List.remove(1);
    Assert::IsTrue(List.head->element == 8);
    List.push_back(3);
    List.remove(3);
    Assert::IsTrue(List.tail->element == 4);
    List.push_back(0);
    List.remove(2);
    buffer = List.head;
    buffer = buffer->next;
    Assert::IsTrue(buffer->element == 0);
}

TEST_METHOD(Testget_size)
{
    List List(5);
    List.push_back(8);
    Assert::IsTrue(List.get_size() == 2);
}
TEST_METHOD(Testclear)
{
    List List(5);
    List.push_back(8);
    List.clear();
    Assert::IsTrue(List.head == NULL && List.tail == NULL);
}
TEST_METHOD(TestisEmpty)
{
    List List;
    List.isEmpty();
    Assert::IsTrue(List.isEmpty() == true);
    Assert::IsTrue(List.get_size() == 0);
}
TEST_METHOD(Testset)
```

```
    }
    TEST_METHOD(Testset)
    {
        List List(5);
        List.push_back(9);
        List.push_back(8);
        int check = 6;
        List.set(6, 1);
        List::Node* buffer = List.head;
        Assert::IsTrue(List.head->element == check);
        List.set(5, 3);
        Assert::IsTrue(List.tail->element == 5);
        List.set(2, 2);
        buffer = buffer->next;
        Assert::IsTrue(buffer->element == 2);
    }
    TEST_METHOD(Testreverse)
    {
        List List(5);
        List.push_back(8);
        List.push_back(3);
        List::Node* buffer = List.head;
        List.reverse();
        Assert::IsTrue(buffer->element == 3);
        buffer = buffer->next;
        Assert::IsTrue(buffer->element == 8);
        buffer = buffer->next;
        Assert::IsTrue(buffer->element == 5);
    }
};
```