**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: «Связный список»**

**Вариант №13**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Приезжих Т.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

**2021**

## Цель

Обучится работе со связными списками, составлению функций для них, освоить использование unit тестов.

### Описание алгоритма

Сначала задаётся класс односвязного списка, в него вкладывается структура узла. На основе этого построены следующие функции.

1. void push\_back(int); добавление элемента в конец списка

2. void push\_front(int); добавление элемента в начало списка

3. void pop\_back(); удаление последнего элемента списка

4. void pop\_front(); удаление первого элемента списка

5. void insert(int, size\_t); добавление элемента списка по индексу (вставка перед элементом, который был ранее доступен по этому индексу)

6. int at(size\_t); получение элемента списка по индексу

7. void remove(size\_t); удаление элемента списка 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 <<

19. int find\_last(List); поиск последнего вхождения другого списка в список

### Оценка временной сложности каждого метода

|  |  |
| --- | --- |
| Метод | Временная сложность |
| push\_back(int) | O(n) |
| push\_front(int) | O(1) |
| void pop\_back() | O(1) |
| void pop\_front() | O(1) |
| void insert(int, size\_t) | O(n) |
| int at(size\_t) | O(n) |
| void remove(size\_t) | O(n) |
| size\_t get\_size() | O(n) |
| void clear() | O(n2) |
| void set(size\_t, int) | 2\*O(n) |
| bool isEmpty() | O(1) |
| int find\_last(List) | O(m\*n) |

### Описание реализованных unit-тестов

1. void push\_back(int); Добавляем в конец списка элемент и сравниваем значение последнего элемента с ожидаемым

2. void push\_front(int); Добавляем в начало списка элемент и сравниваем значение первого элемента с ожидаемым

3. void pop\_back(); Создаём список из нескольких элементов, вызываем функцию, сравниваем значение элемента с ожидаемым

4. void pop\_front(); Создаём список из нескольких элементов, вызываем функцию, сравниваем значение элемента с ожидаемым

5. void insert(int, size\_t); Добавляем элемент с индексом 2 (например), затем с помощью функции at(size\_t); проверяем значение элемента с индексом 2

6. int at(size\_t); Создаем список, добаляем в него элемент и сравниваем его значение с ожидаемым

7. void remove(size\_t); Удаляем элемент с индексом 1 (например), сравниваем значение элемента с ожидаемым

8. size\_t get\_size(); Добавляем в список 10 элементов (например), вызываем функцию и сравниваем её значение с ожидаемым

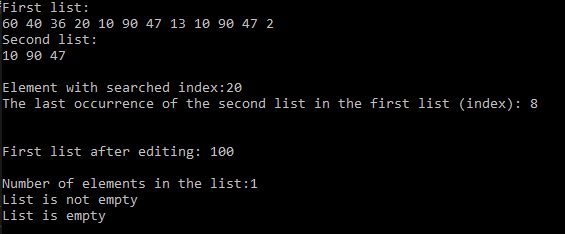
9. void clear(); Создаём список из нескольких элементов, вызываем функцию, проверяем список на пустоту

10. void set(size\_t, int);

11. bool isEmpty(); Создаём список с одним элементом, затем удаляем элемент и проверяем список на пустоту

19. int find\_last(List); Создаём первый список с множеством элементов, повторяем вхождение элементов (например 1 2 3) несколько раз, затем создаём второй список с элементами (например 1 2 3), сравниваем полученный индекс с ожидаемым

### Пример работы



### Листинг

Lab1.h

#pragma once

#include <iostream>

using namespace std;

class List {

private:

struct Node {

int data;

Node\* next;

Node(int input) {

data = input;

next = nullptr;

}

int getData() {

return data;

}

Node\* getNext() {

return next;

}

void setNext(Node\* newnext) {

next = newnext;

}

};

Node\* head;

void push\_front(Node\* temp) {

temp->next = head;

head = temp;

}

void push\_back(Node\* temp) {

Node\* p = head;

for (size\_t i = 0; i < get\_size() - 1; i++) {

p = p->next;

}

p->next = temp;

}

void push(Node\* temp, Node\* prev, Node\* prev\_plus\_one) {

temp->next = prev\_plus\_one;

prev->next = temp;

temp = prev;

}

void pop\_front\_() {

Node\* p = head;

head = head->next;

delete p;

}

void pop\_back\_() {

Node\* p = head;

head = head->next;

if (head == nullptr)

delete p;

}

size\_t get\_size\_() {

Node\* p = head;

size\_t count = 0;

while (p != nullptr) {

count++;

p = p->next;

}

return count;

}

void remove\_(size\_t number) {

Node\* p = head;

Node\* cur = head;

Node\* prev = NULL;

bool found = false;

for (size\_t i = 0; i < get\_size() - 1; i++) {

if (i == number - 1) {

prev = cur;

cur = cur->getNext();

}

else {

if (i < number)

cur = cur->getNext();

}

}

if (prev == NULL) {

head = cur->getNext();

}

else {

prev->setNext(cur->getNext());

}

}

int at\_(size\_t number) {

Node\* p = head;

for (size\_t i = 0; i < get\_size(); i++) {

if (i == number) {

return p->getData();

}

else {

p = p->next;

}

}

return false;

}

void insert\_(int data, size\_t number) {

Node\* cur = head;

Node\* prev = NULL;

bool found = false;

for (size\_t i = 0; i < get\_size() - 1; i++) {

if (i == number - 1) {

prev = cur;

cur = cur->getNext();

}

else {

if (i < number)

cur = cur->getNext();

}

}

push(new Node(data), prev, cur);

}

int find\_last\_(List\* second\_list) {

Node\* p1 = head;

Node\* p2 = second\_list->head;

size\_t size1 = get\_size();

size\_t size2 = second\_list->get\_size();

int index = -1;

for (size\_t i = 0; i < size1; i++) {

int n1 = this->at(i);

int n2 = second\_list->at(0);

if (n1 == n2) {

if (!(i + (size2 - 1) < size1)) break;

bool is\_sublist = true;

for (size\_t j = 0; j < size2 && i + j < size1; j++) {

n1 = this->at(i + j);

n2 = second\_list->at(j);

if (n1 != n2) is\_sublist = false;

}

if (is\_sublist) index = i;

}

}

return index;

}

public:

List() {

head = nullptr;

}

List(int data) {

head = new Node(data);

}

~List() {

while (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

void push\_front(int data) { //pushes the element in the front

if (head == nullptr) {

head = new Node(data);

}

else

push\_front(new Node(data));

}

void push\_back(int data) { //pushes the element in the back

if (head == nullptr) {

head = new Node(data);

}

else

push\_back(new Node(data));

}

void pop\_back() { //deletes the last element

if (head == nullptr) return;

else pop\_back\_();

}

void pop\_front() { //deletes the first element

if (head == nullptr) return;

else pop\_front\_();

}

void insert(int data, size\_t number) { //inserts an element (data) on index (number)

if (head == nullptr) return;

else insert\_(data, number);

}

int at(size\_t number) { //Finds an element with index (number)

if (head == nullptr) return 0;

else return at\_(number);

}

void remove(size\_t number) { //Removes an element with index (number)

if ((head == nullptr) || (number > get\_size()) || (number < 0)) return;

else remove\_(number);

}

size\_t get\_size() { //Gets the size of the list

if (head == nullptr) return 0;

else return get\_size\_();

}

void clear() { //Removes all the elements in the list

if (head == nullptr) return;

else while (head != nullptr) pop\_front();

}

void set(size\_t number, int data) { //Removes an element with index (number) and puts another element (data) on index

if (head == nullptr) return;

else {

remove(number);

insert(data, number);

}

}

bool isEmpty() { //Finds out if the list is empty

if (head == nullptr) return true;

else return false;

}

int find\_last(List\* second\_list) { //Searches for the last occurrence of another list (second\_list) in the list

if (head == nullptr) return -1;

else {

return find\_last\_(second\_list);

}

}

friend ostream& operator<< (ostream& out, List& l);

};

ostream& operator<< (ostream& out, List& l) {

List::Node\* p = l.head;

while (p != nullptr) {

cout << p->data << " ";

p = p->next;

}

return out;

}

Lab1.cpp

#include <iostream>

#include <locale>

#include "Lab1.h"

using namespace std;

int main()

{

//Testing if everything is working fine

List\* p = new List(10);

List\* r = new List(10);

r->push\_back(90);

r->push\_back(47);

p->push\_front(20);

p->push\_back(47);

p->push\_front(30);

p->push\_back(13);

p->push\_front(40);

p->push\_front(50);

p->push\_front(60);

p->push\_back(10);

p->push\_back(90);

p->push\_back(47);

p->push\_back(2);

p->remove(1);

p->insert(90, 5);

p->set(2, 36);

cout << "First list:\n" << \*p << "\n";

cout << "Second list:\n" << \*r << "\n";

cout << "\nElement with searched index:" << p->at(3) << "\n";

if (p->find\_last(r) == -1) cout << "First list do not contain second list\n\n";

else cout << "The last occurrence of the second list in the first list (index): " << p->find\_last(r) << "\n\n";

p->clear();

p->push\_back(100);

cout << "\nFirst list after editing: " << \*p << "\n";

cout << "\nNumber of elements in the list:" << p->get\_size() << "\n";

if (p->isEmpty()) cout << "List is empty\n"; else cout << "List is not empty\n";

p->pop\_front();

p->pop\_front();

p->pop\_back();

p->pop\_back();

if (p->isEmpty()) cout << "List is empty\n"; else cout << "List is not empty\n";

}

UnitTest1.cpp

#include "pch.h"

#include "CppUnitTest.h"

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

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Tests

{

TEST\_CLASS(Tests)

{

public:

TEST\_METHOD(AddingElements)

{

List\* p = new List(6); //List: 6

Assert::AreEqual(6, p->at(0));

p->push\_back(11); //List: 6 11

Assert::AreEqual(11, p->at(1));

p->push\_front(17); //List: 6 11 17

Assert::AreEqual(17, p->at(0));

p->insert(13, 2); //List: 6 11 13 17

Assert::AreEqual(13, p->at(2));

}

TEST\_METHOD(DeletingElements) {

List\* p = new List(6); //List: 6

p->pop\_back(); //List: nothing

Assert::IsTrue(p->isEmpty());

p->push\_back(1);

p->push\_back(2);

p->push\_back(3); //List: 1 2 3

Assert::IsFalse(p->isEmpty());

p->pop\_front(); //List: 2 3

p->push\_back(4);

p->push\_back(5); //List: 2 3 4 5

p->remove(1); //List: 2 4 5

Assert::AreEqual(4, p->at(1));

p->clear(); //List: nothing

Assert::IsTrue(p->isEmpty());

}

TEST\_METHOD(SetElement) {

List\* p = new List(6); //List: 6

p->push\_back(1);

p->push\_back(2); //List: 6 1 2

p->set(1, 4); //List: 6 4 2

Assert::AreEqual(4, p->at(1));

}

TEST\_METHOD(GetSize) {

List\* p = new List(6); //List: 6

for (size\_t i = 0; i < 9; i++)

p->push\_back(1); //List: 6 1 1 1 1 1 1 1 1 1 (10 numbers)

Assert::AreEqual((size\_t)10, p->get\_size());

}

TEST\_METHOD(FindLastElement) {

List\* p = new List();

p->push\_back(1);

p->push\_back(2);

p->push\_back(3); //List: 1 2 3

for (int i = 0; i < 5; i++) p->push\_back(i + 10); //List: 1 2 3 10 11 12 13 14

p->push\_back(1);

p->push\_back(2);

p->push\_back(3);

for (int i = 0; i < 5; i++) p->push\_back(i + 7); //List: 1 2 3 10 11 12 13 14 1 2 3 7 8 9 10 11

p->push\_back(1);

p->push\_back(2);

p->push\_back(3);

for (int i = 0; i < 5; i++) p->push\_back(i + 8); //List: 1 2 3 10 11 12 13 14 1 2 3 7 8 9 10 11 1 <--(16th element by index) 2 3 8 9 10 11 12

List\* sub\_p = new List();

sub\_p->push\_back(1);

sub\_p->push\_back(2);

sub\_p->push\_back(3); //List2: 1 2 3

Assert::AreEqual(16, p->find\_last(sub\_p));

}

};

}