МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

ЭЛЕКТРОТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА «ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И АВТОМАТИЗИРОВАННЫХ СИСТЕМ»

**ОТЧЁТ**

**«ЛАБОРАТОРНАЯ №11: ОДНОСВЯЗНЫЙ И ДВУСВЯЗНЫЙ СПИСОК»**

Дисциплина: «Программирование»

Выполнил:

Студент группы ИВТ-21-2б

Безух Владимир Сергеевич

Проверил:

Доцент кафедры ИТАС

Полякова Ольга Андреевна

Пермь, 2022

Содержание

[1. Постановка задачи 3](#_Toc97829619)

[2. Анализ задачи 4](#_Toc97829620)

[3. Описание переменных 5](#_Toc97829621)

[4. Исходный код 6](#_Toc97829622)

[5. Анализ результатов 18](#_Toc97829623)

# Постановка задачи

Односвязный список чисел: удалить из списка все элементы с чётными значениями.

Двусвязный список строк: добавить в список узлы по нечётным индексам.

# 

# Анализ задачи

Обе задачи — тривиальны. Достаточно организовать свои структуры данных для дальнейшей работы.

# 

# Описание переменных

**SinglyList<int> list;** — односвязный список целых чисел.

**DoublyList<std::string> list;** — двусвязный список строк.

# Исходный код

#include<string>

#include<iostream>

#include<vld.h>

template <typename T>

class SinglyList {

private:

struct Node;

public:

class Iterator {

public:

Iterator() : current\_node(nullptr) {}

Iterator(Node\* node)

: current\_node(node) {}

T operator\*() const { return current\_node->data; }

bool operator==(const Iterator& right) const {

return current\_node == right.current\_node;

}

bool operator!=(const Iterator& right) const {

return !(\*this == right);

}

Iterator& operator++() {

if (current\_node != nullptr)

current\_node = current\_node->pointer\_to\_next\_node;

return \*this;

}

private:

Node\* current\_node;

};

public:

SinglyList();

~SinglyList();

size\_t size() const;

void pushBack(const T& data);

void pushFront(const T& data);

void popBack();

void popFront();

void insert(int index, const T& data);

void remove(int index);

void clear();

Iterator begin() const;

Iterator end() const;

private:

struct Node {

Node(T data = T(), Node\* pointer\_to\_next\_node = nullptr)

: data(data), pointer\_to\_next\_node(pointer\_to\_next\_node) {}

Node(const Node& copy)

: data(copy.data), pointer\_to\_next\_node(copy.pointer\_to\_next\_node) {}

Node& operator=(const Node& right) {

if (this != &right) {

data = right.data;

pointer\_to\_next\_node = right.pointer\_to\_next\_node;

}

return \*this;

}

T data;

Node\* pointer\_to\_next\_node;

};

void pushFirstNode(Node\* node);

void pushBackNode(Node\* node);

void pushFrontNode(Node\* node);

void insertRightToNode(Node\* current\_node, Node\* insert\_node);

void popFirstNode();

void popBackNode();

void popFrontNode();

void removeNextNode(Node\* node);

size\_t normalizeIndex(int index) const;

Node\* findNode(const size\_t& index) const;

size\_t list\_size;

Node\* head\_node;

Node\* tail\_node;

};

template<typename T>

SinglyList<T>::SinglyList()

: list\_size(size\_t{0}), head\_node(nullptr), tail\_node(nullptr) {}

template<typename T>

SinglyList<T>::~SinglyList()

{

clear();

}

template<typename T>

size\_t SinglyList<T>::size() const

{

return list\_size;

}

template<typename T>

void SinglyList<T>::pushBack(const T& data)

{

Node\* new\_node = new Node(data);

list\_size ? pushBackNode(new\_node) : pushFirstNode(new\_node);

++list\_size;

}

template<typename T>

void SinglyList<T>::pushFront(const T& data)

{

Node\* new\_node = new Node(data);

list\_size ? pushFrontNode(new\_node) : pushFirstNode(new\_node);

++list\_size;

}

template<typename T>

void SinglyList<T>::popBack()

{

if (list\_size == size\_t{0}) return;

Node\* remove\_node = tail\_node;

(list\_size == size\_t{1}) ? popFirstNode() : popBackNode();

delete remove\_node;

--list\_size;

}

template<typename T>

void SinglyList<T>::popFront()

{

if (list\_size == size\_t{0}) return;

Node\* remove\_node = head\_node;

(list\_size == size\_t{1}) ? popFirstNode() : popFrontNode();

delete remove\_node;

--list\_size;

}

template<typename T>

void SinglyList<T>::insert(int index, const T& data)

{

Node\* new\_node = new Node(data);

if (list\_size == size\_t{0}) { pushFirstNode(new\_node); ++list\_size; return; }

size\_t normalize\_index = normalizeIndex(index);

if (normalize\_index == size\_t{0}) pushFrontNode(new\_node);

else {

Node\* found\_node = findNode(--normalize\_index);

insertRightToNode(found\_node, new\_node);

}

++list\_size;

}

template<typename T>

void SinglyList<T>::remove(int index)

{

if (list\_size == size\_t{0}) { return; }

Node\* remove\_node = head\_node;

if (list\_size == size\_t{1}) { popFirstNode(); delete remove\_node; --list\_size; return; }

size\_t normalize\_index = normalizeIndex(index);

if (normalize\_index == size\_t{0}) popFrontNode();

else if (normalize\_index == list\_size - size\_t{1}) {

remove\_node = tail\_node; popBackNode();

}

else {

Node\* node = findNode(--normalize\_index);

remove\_node = node->pointer\_to\_next\_node;

removeNextNode(node);

}

delete remove\_node;

--list\_size;

}

template<typename T>

void SinglyList<T>::clear()

{

if (list\_size == size\_t{0}) return;

Node\* remove;

Node\* next\_node = head\_node;

while (list\_size) {

remove = next\_node;

next\_node = next\_node->pointer\_to\_next\_node;

delete remove;

--list\_size;

}

head\_node = nullptr;

tail\_node = nullptr;

}

template<typename T>

typename SinglyList<T>::Iterator SinglyList<T>::begin() const

{

return Iterator(head\_node);

}

template<typename T>

typename SinglyList<T>::Iterator SinglyList<T>::end() const

{

return Iterator(nullptr);

}

template<typename T>

void SinglyList<T>::pushFirstNode(Node\* node)

{

head\_node = node;

tail\_node = node;

}

template<typename T>

void SinglyList<T>::pushBackNode(Node\* node)

{

tail\_node->pointer\_to\_next\_node = node;

tail\_node = node;

}

template<typename T>

void SinglyList<T>::pushFrontNode(Node\* node)

{

node->pointer\_to\_next\_node = head\_node;

head\_node = node;

}

template<typename T>

void SinglyList<T>::insertRightToNode(Node\* current\_node, Node\* insert\_node)

{

insert\_node->pointer\_to\_next\_node = current\_node->pointer\_to\_next\_node;

current\_node->pointer\_to\_next\_node = insert\_node;

}

template<typename T>

void SinglyList<T>::popFirstNode()

{

head\_node = nullptr;

tail\_node = nullptr;

}

template<typename T>

void SinglyList<T>::popBackNode()

{

Node\* node = findNode(list\_size - size\_t{2});

removeNextNode(node);

tail\_node = node;

}

template<typename T>

void SinglyList<T>::popFrontNode()

{

head\_node = head\_node->pointer\_to\_next\_node;

}

template<typename T>

void SinglyList<T>::removeNextNode(Node\* node)

{

node->pointer\_to\_next\_node = node->pointer\_to\_next\_node->pointer\_to\_next\_node;

}

template<typename T>

size\_t SinglyList<T>::normalizeIndex(int index) const

{

int temp\_size = static\_cast<int>(list\_size);

index %= temp\_size; if (index < 0) index += temp\_size;

return static\_cast<size\_t>(index);

}

template<typename T>

typename SinglyList<T>::Node\* SinglyList<T>::findNode(const size\_t& index) const

{

if (index == size\_t{0}) return head\_node;

if (index == list\_size - size\_t{1}) return tail\_node;

Node\* node = head\_node;

for (size\_t counter = 0; counter != index; ++counter)

node = node->pointer\_to\_next\_node;

return node;

}

template <typename T>

class DoublyList {

private:

struct Node;

public:

class Iterator {

public:

Iterator() : current\_node(nullptr) {}

Iterator(Node\* node)

: current\_node(node) {}

T operator\*() const { return current\_node->data; }

bool operator==(const Iterator& right) const {

return current\_node == right.current\_node;

}

bool operator!=(const Iterator& right) const {

return !(\*this == right);

}

Iterator& operator++() {

if (current\_node != nullptr)

current\_node = current\_node->pointer\_to\_next\_node;

return \*this;

}

Iterator& operator--() {

if (current\_node != nullptr)

current\_node = current\_node->pointer\_to\_prev\_node;

return \*this;

}

private:

Node\* current\_node;

};

public:

DoublyList();

~DoublyList();

size\_t size() const;

void pushBack(const T& data);

void pushFront(const T& data);

void popBack();

void popFront();

void insert(int index, const T& data);

void remove(int index);

void clear();

Iterator begin() const;

Iterator end() const;

private:

struct Node {

Node(T data = T(), Node\* pointer\_to\_prev\_node = nullptr, Node\* pointer\_to\_next\_node = nullptr)

: data(data), pointer\_to\_prev\_node(pointer\_to\_prev\_node), pointer\_to\_next\_node(pointer\_to\_next\_node) {}

Node(const Node& copy)

: data(copy.data), pointer\_to\_prev\_node(copy.pointer\_to\_prev\_node), pointer\_to\_next\_node(copy.pointer\_to\_next\_node) {}

Node& operator=(const Node& right) {

if (this != &right) {

data = right.data;

pointer\_to\_prev\_node = right.pointer\_to\_prev\_node;

pointer\_to\_next\_node = right.pointer\_to\_next\_node;

}

return \*this;

}

T data;

Node\* pointer\_to\_prev\_node;

Node\* pointer\_to\_next\_node;

};

void pushFirstNode(Node\* node);

void pushBackNode(Node\* node);

void pushFrontNode(Node\* node);

void insertNode(Node\* current\_node, Node\* insert\_node);

void popFirstNode();

void popBackNode();

void popFrontNode();

void removeNode(Node\* node);

size\_t normalizeIndex(int index) const;

Node\* findNode(const size\_t& index) const;

size\_t list\_size;

Node\* head\_node;

Node\* tail\_node;

};

template<typename T>

DoublyList<T>::DoublyList()

: list\_size(size\_t{0}), head\_node(nullptr), tail\_node(nullptr) {}

template<typename T>

DoublyList<T>::~DoublyList()

{

clear();

}

template<typename T>

size\_t DoublyList<T>::size() const

{

return list\_size;

}

template<typename T>

void DoublyList<T>::pushBack(const T& data)

{

Node\* new\_node = new Node(data);

list\_size ? pushBackNode(new\_node) : pushFirstNode(new\_node);

++list\_size;

}

template<typename T>

void DoublyList<T>::pushFront(const T& data)

{

Node\* new\_node = new Node(data);

list\_size ? pushFrontNode(new\_node) : pushFirstNode(new\_node);

++list\_size;

}

template<typename T>

void DoublyList<T>::popBack()

{

if (list\_size == size\_t{0}) return;

Node\* remove\_node = tail\_node;

(list\_size == size\_t{1}) ? popFirstNode() : popBackNode();

delete remove\_node;

--list\_size;

}

template<typename T>

void DoublyList<T>::popFront()

{

if (list\_size == size\_t{0}) return;

Node\* remove\_node = head\_node;

(list\_size == size\_t{1}) ? popFirstNode() : popFrontNode();

delete remove\_node;

--list\_size;

}

template<typename T>

void DoublyList<T>::insert(int index, const T& data)

{

Node\* new\_node = new Node(data);

if (list\_size == size\_t{0}) { pushFirstNode(new\_node); ++list\_size; return; }

size\_t normalize\_index = normalizeIndex(index);

if (normalize\_index == size\_t{0}) pushFrontNode(new\_node);

else {

Node\* found\_node = findNode(normalize\_index);

insertNode(found\_node, new\_node);

}

++list\_size;

}

template<typename T>

void DoublyList<T>::remove(int index)

{

if (list\_size == size\_t{0}) { return; }

Node\* remove\_node = head\_node;

if (list\_size == size\_t{1}) { popFirstNode(); delete remove\_node; --list\_size; return; }

size\_t normalize\_index = normalizeIndex(index);

if (normalize\_index == size\_t{0}) popFrontNode();

else if (normalize\_index == list\_size - size\_t{1}) {

remove\_node = tail\_node; popBackNode();

}

else {

remove\_node = findNode(normalize\_index);

removeNode(remove\_node);

}

delete remove\_node;

--list\_size;

}

template<typename T>

void DoublyList<T>::clear()

{

if (list\_size == size\_t{ 0 }) return;

Node\* remove;

Node\* next\_node = head\_node;

while (list\_size) {

remove = next\_node;

next\_node = next\_node->pointer\_to\_next\_node;

delete remove;

--list\_size;

}

head\_node = nullptr;

tail\_node = nullptr;

}

template<typename T>

typename DoublyList<T>::Iterator DoublyList<T>::begin() const

{

return Iterator(head\_node);

}

template<typename T>

typename DoublyList<T>::Iterator DoublyList<T>::end() const

{

return Iterator(nullptr);

}

template<typename T>

void DoublyList<T>::pushFirstNode(Node\* node)

{

head\_node = node;

tail\_node = node;

}

template<typename T>

void DoublyList<T>::pushBackNode(Node\* node)

{

tail\_node->pointer\_to\_next\_node = node;

node->pointer\_to\_prev\_node = tail\_node;

tail\_node = node;

}

template<typename T>

void DoublyList<T>::pushFrontNode(Node\* node)

{

node->pointer\_to\_next\_node = head\_node;

head\_node->pointer\_to\_prev\_node = node;

head\_node = node;

}

template<typename T>

void DoublyList<T>::insertNode(Node\* current\_node, Node\* insert\_node)

{

insert\_node->pointer\_to\_prev\_node = current\_node->pointer\_to\_prev\_node;

insert\_node->pointer\_to\_next\_node = current\_node;

current\_node->pointer\_to\_prev\_node = insert\_node;

insert\_node->pointer\_to\_prev\_node->pointer\_to\_next\_node = insert\_node;

}

template<typename T>

void DoublyList<T>::popFirstNode()

{

head\_node = nullptr;

tail\_node = nullptr;

}

template<typename T>

void DoublyList<T>::popBackNode()

{

tail\_node->pointer\_to\_prev\_node->pointer\_to\_next\_node = nullptr;

tail\_node = tail\_node->pointer\_to\_prev\_node;

}

template<typename T>

void DoublyList<T>::popFrontNode()

{

head\_node->pointer\_to\_next\_node->pointer\_to\_prev\_node = nullptr;

head\_node = head\_node->pointer\_to\_next\_node;

}

template<typename T>

void DoublyList<T>::removeNode(Node\* node)

{

node->pointer\_to\_prev\_node->pointer\_to\_next\_node = node->pointer\_to\_next\_node;

node->pointer\_to\_next\_node->pointer\_to\_prev\_node = node->pointer\_to\_prev\_node;

}

template<typename T>

size\_t DoublyList<T>::normalizeIndex(int index) const

{

int temp\_size = static\_cast<int>(list\_size);

index %= temp\_size; if (index < 0) index += temp\_size;

return static\_cast<size\_t>(index);

}

template<typename T>

typename DoublyList<T>::Node\* DoublyList<T>::findNode(const size\_t& index) const

{

if (index == size\_t{0}) return head\_node;

size\_t last = list\_size - size\_t{1};

if (index == last) return tail\_node;

Node\* node;

size\_t from\_tail = last - index;

if (index < from\_tail)

{

node = head\_node;

for (size\_t counter = 0; counter != index; ++counter)

node = node->pointer\_to\_next\_node;

}

else {

node = tail\_node;

for (size\_t counter = 0; counter != from\_tail; ++counter)

node = node->pointer\_to\_prev\_node;

}

return node;

}

template<class T>

void printList(const T& list)

{

for (auto it = list.begin(); it != list.end(); ++it)

std::cout << \*it << ' ';

std::cout << '\n';

}

void firstTask()

{

SinglyList<int> list;

for (size\_t i = 0; i != 10; ++i)

list.pushBack(i \* i);

printList(list);

size\_t i = 0;

for (auto it = list.begin(); it != list.end();) {

if (\*it % 2 == 0) { ++it; list.remove(i); }

else { ++it; ++i; }

}

printList(list);

}

void secondTask()

{

DoublyList<std::string> list;

list.pushBack("str2"); list.pushBack("str4"); list.pushBack("str6");

printList(list);

list.insert(0, "str1"); list.insert(2, "str3"); list.insert(4, "str5");

printList(list);

}

int main()

{

firstTask();

secondTask();

}

# Анализ результатов

Результаты работы программы (рис. 1).

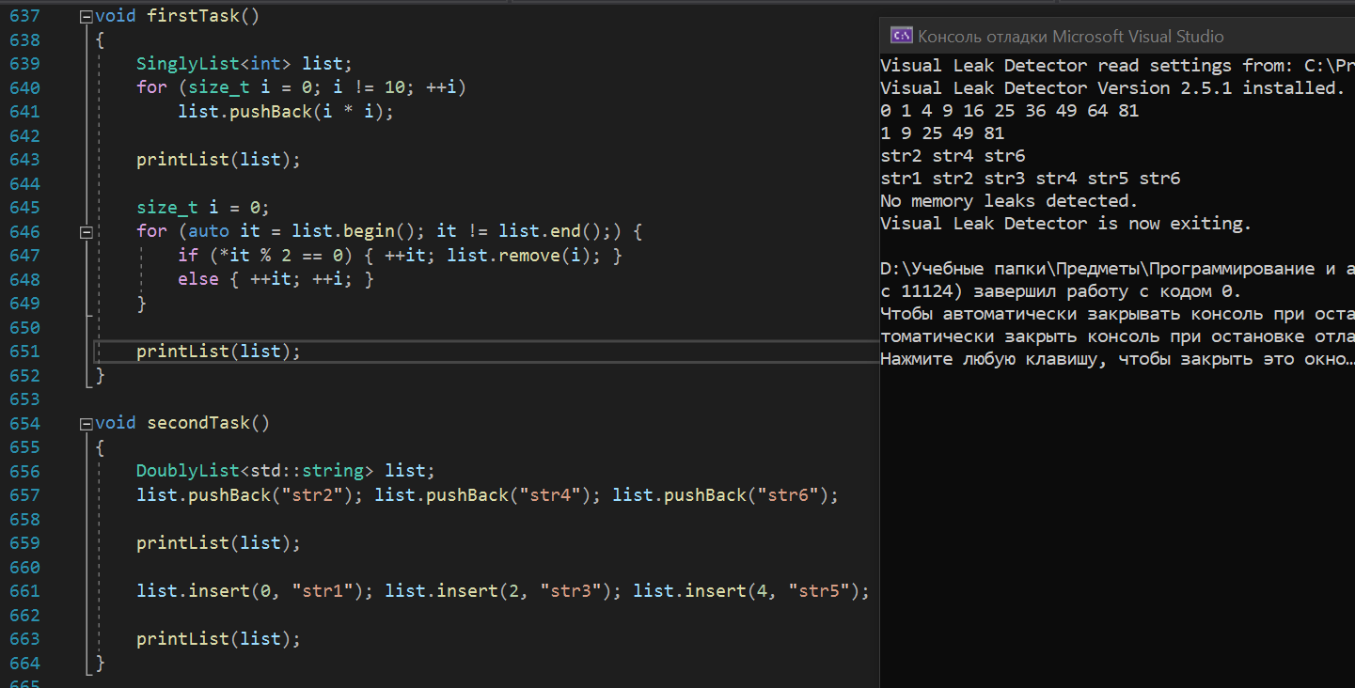


Рисунок — Результаты