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

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

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

2. Анализ задачи

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

3. Описание переменных

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

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

4. 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;
}
```

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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();
}
```

5. Анализ результатов

Результаты работы программы (рис. 1).

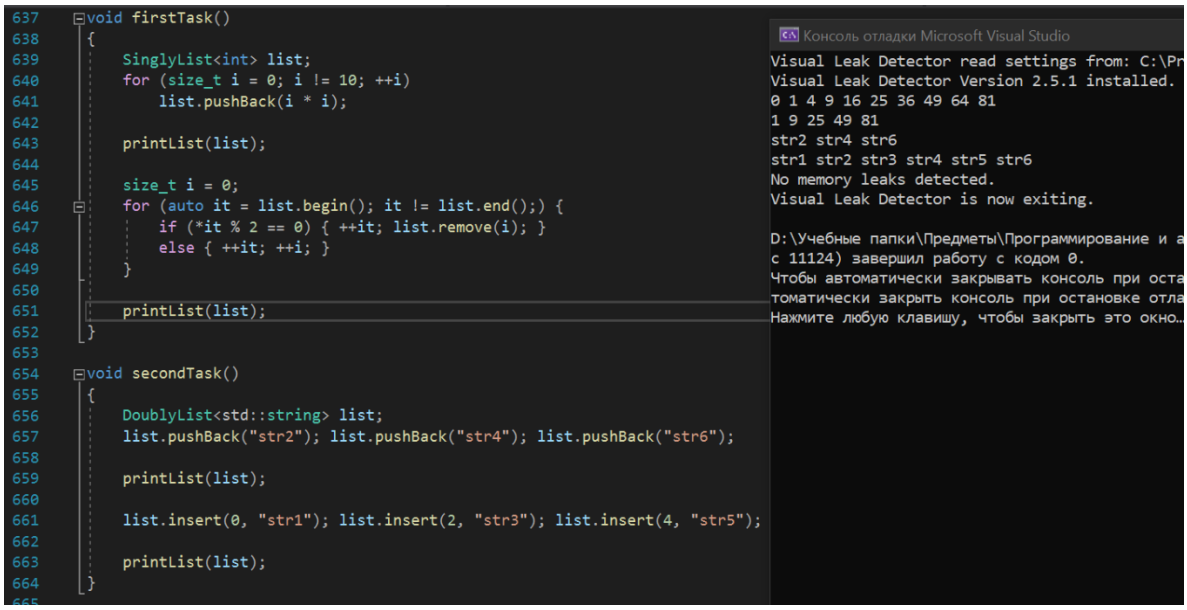

Рисунок 1 — Результаты