Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Выполнил студент группы КС-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 2](#_Toc63548272)

[Описание метода/модели. 2](#_Toc63548273)

[Выполнение задачи. 2](#_Toc63548274)

[Заключение. 2](#_Toc63548275)

# Описание задачи.

Целью лабораторной работы была реализация двусвязного списка на языке C++ с использованием шаблонного подхода. Требовалось реализовать следующие операции:

* Добавление элемента в начало, в конец и в произвольное место списка.
* Удаление элемента из списка.
* Реализация итератора для обхода списка.
* Проверка на пустоту и подсчет количества элементов.

**Задание: Вариант 1**

1. Написать свою реализацию двусвязного списка

* Добавление элемента в начало, в конец, в произвольное место
* Удаление элемента по из списка

В рамках лабораторной работы необходимо изучить и реализовать одну из трёх структур(двухсвязный список, стек, очередь), в соответствии со своим вариантом, при этом, все структуры должны:

* Использовать шаблонный подход, обеспечивая работу контейнера с произвольными данными.
* Реализовывать свой итератор предоставляющий стандартный для языка механизм работы с ним(для С++ это операции ++ и операция !=, для python это )
* Обеспечивать работу стандартных библиотек и конструкции for each если она есть в языке, если их нет, то реализовать собственную функцию использующую итератор.
* Проверку на пустоту и подсчет количества элементов.

Для демонстрации работы структуры необходимо создать набор тестов(под тестом понимается функция, которая создаёт структуру, проводит операцию или операции над структурой и удаляет структуру):

* заполнение контейнера 1000 целыми числами в диапазоне от -1000 до 1000 и подсчет их суммы, среднего, минимального и максимального.
* Провести проверку работы операций вставки и изъятия элементов на коллекции из 10 строковых элементов.
* заполнение контейнера 100 структур содержащих фамилию, имя, отчество и дату рождения(от 01.01.1980 до 01.01.2020) значения каждого поля генерируются случайно из набора заранее заданных. После заполнение необходимо найти всех людей младше 20 лет и старше 30 и создать новые структуры содержащие результат фильтрации, проверить выполнение на правильность подсчётом кол-ва элементов не подходящих под условие в новых структурах. Тесты:

Список

* Перемешать все элементы
* Выполнить серию тестирования сортировки выбором на реализованном списке и сравнить производительность с полученной на массиве.

# Описание метода/модели.

**Двусвязный список**

Двусвязный список — это структура данных, состоящая из узлов, каждый из которых содержит:

* Данные (произвольного типа).
* Указатель на предыдущий узел.
* Указатель на следующий узел.

**Преимущества двусвязного списка:**

* Динамическое изменение размера.
* Эффективное добавление и удаление элементов в начало и конец.
* Возможность обхода в обоих направлениях.

**Недостатки двусвязного списка:**

* Больший расход памяти по сравнению с односвязным списком (из-за хранения двух указателей в каждом узле).
* Медленный доступ к элементам по индексу (O(n)).

# 

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

#include <iostream>

#include <random>

#include <string>

#include <vector>

#include <algorithm>

#include <ctime>

#include <cstdlib>

#include <chrono>

#include <random>

#include <iomanip>

#include <numeric>

// Шаблонный класс двусвязного списка

template <typename T> //шаблонный класс позволяет использовать список для любого типа данных

class DoublyLinkedList { //шаблонный класс

private: //для избегания случайного изменения внутренних данных извне

struct Node { //Узел списка

T data; //данные

Node\* prev; //указатель на предыдущий узел

Node\* next; //указатель на следующий узел

Node(const T& data) : data(data), prev(nullptr), next(nullptr) {}

}; // конструктор структуры Node. Он принимает данные (data) и инициализирует поля структуры.

Node\* head; //указатель на первый узел списка

Node\* tail; //указатель на последний узел списка

size\_t size; //количество элементов в списке

//итератор

public: //так как пользователь может обходить список

// Итератор для двусвязного списка

class Iterator { //внутренний класс для обхода списка

private:

Node\* current; //указатель на текущий узел

public:

Iterator(Node\* node) : current(node) {} // Конструктор. Принимает указатель на узел и инициализирует поле current этим указателем.

T& operator\*() { return current->data; } //возвращает данные текущего узла

Iterator& operator++() { //переходит к следующему узлу

current = current->next;

return \*this;

}

bool operator!=(const Iterator& other) const { //проверяет не равен ли текущий узел другому узлу

return current != other.current;

}

};

//конструктор (инициализирует начало/конец/размер нулевыми значениями)

DoublyLinkedList() : head(nullptr), tail(nullptr), size(0) {}

//деструктор (освобождает памятьБ удаляя все узлы списка)

~DoublyLinkedList() {

while (head) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

//Метод добавляет новый элемент в начало списка.

void push\_front(const T& data) {

Node\* newNode = new Node(data); //Создает новый узел с данными data

if (!head) { //Проверяет, пуст ли список. Если список пуст:

head = tail = newNode; //Устанавливает head и tail на новый узел, так как он становится единственным элементом списка.

} else { //Если список не пуст:

newNode->next = head; // Устанавливает указатель next нового узла на текущий первый узел (head).

head->prev = newNode; //станавливает указатель prev текущего первого узла на новый узел

head = newNode; // Перемещает head на новый узел, так как он теперь первый.

}

size++; //Увеличивает счетчик элементов списка.

}

void push\_back(const T& data) {

Node\* newNode = new Node(data);

if (!tail) {

head = tail = newNode;

} else {

newNode->prev = tail;

tail->next = newNode;

tail = newNode;

}

size++;

}

void insert(const T& data, size\_t position) {

if (position > size) return; // если позиция больше размера списка, функция завершается.

if (position == 0) { //если позиция равна 0, вызывается push\_front

push\_front(data);

} else if (position == size) { // если позиция равна размеру списка, вызывается push\_back

push\_back(data);

} else { //Вставка в произвольное место:

Node\* newNode = new Node(data); //Создаётся новый узел

Node\* current = head;

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

current = current->next;

}

//Устанавливаются указатели нового узла:

newNode->next = current->next;

newNode->prev = current;

//Обновляются указатели соседних узлов:

current->next->prev = newNode;

current->next = newNode;

size++;

}

}

void remove(size\_t position) {

if (position >= size) return; //если позиция больше или равна размеру списка, функция завершается.

Node\* toDelete;

//Удаление первого элемента:

if (position == 0) {

toDelete = head; //удаляемый узел — это голова

head = head->next; //голова перемещается на следующий узел.

if (head) head->prev = nullptr;

else tail = nullptr;

//Удаление последнего элемента:

} else if (position == size - 1) {

toDelete = tail;

tail = tail->prev;

tail->next = nullptr;

} else { //Удаление из середины списка

Node\* current = head;

for (size\_t i = 0; i < position; ++i) { // следующий узел предыдущего узла теперь указывает на следующий узел удаляемого.

current = current->next;

}

toDelete = current;

current->prev->next = current->next;

current->next->prev = current->prev;

}

delete toDelete;

size--;

}

// bool empty() const { return size == 0; } // Проверяет, пуст ли список.

size\_t getSize() const { return size; } //Возвращает количество элементов в списке.

//Возвращают итераторы на начало и конец списка.

Iterator begin() { return Iterator(head); }

Iterator end() { return Iterator(nullptr); }

//Выводит все элементы списка на экран.

void print() const {

Node\* current = head;

while (current) {

std::cout << current->data << " "; // Выводит данные текущего узла на экран.

current = current->next; //Перемещает указатель current на следующий узел списка.

}

std::cout << "\n";

}

void shuffle() {

std::vector<T> elements; // Вектор для временного хранения элементов списка

for (auto& item : \*this) { // Копируем элементы списка в вектор

elements.push\_back(item);

}

std::shuffle(elements.begin(), elements.end(),

std::mt19937(std::random\_device()())); // Перемешиваем элементы вектора

Node\* current = head; // Указатель на текущий узел списка

for (auto& item : elements) { // Копируем перемешанные элементы обратно в список

current->data = item;

current = current->next;

}

}

void selectionSort(size\_t& passCount, size\_t& swapCount) {

passCount = 0;

swapCount = 0;

Node\* current = head;

while (current != nullptr) {

Node\* minNode = current;

Node\* nextNode = current->next;

// Поиск минимального элемента в оставшейся части списка

while (nextNode != nullptr) {

passCount++; // Увеличиваем счетчик проходов

if (nextNode->data < minNode->data) {

minNode = nextNode;

}

nextNode = nextNode->next;

}

// Если минимальный элемент не равен текущему, меняем их местами

if (minNode != current) {

// Обмен данными между current и minNode

std::swap(current->data, minNode->data);

swapCount++; // Увеличиваем счетчик обменов

}

// Переходим к следующему узлу

current = current->next;

}

}

};

// Генерация случайных чисел с плавающей запятой

double generateRandomDouble(double min, double max) {

static std::random\_device rd;

static std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dis(min, max);

return dis(gen);

}

// Тестирование сортировки

void testSortingPerformance() {

const std::vector<size\_t> sizes = {1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000};

const int seriesCount = 8;

const int attemptsPerSeries = 20;

for (size\_t size : sizes) {

std::vector<double> listTimes;

size\_t listPassCount = 0, listSwapCount = 0;

for (int series = 0; series < seriesCount; ++series) {

for (int attempt = 0; attempt < attemptsPerSeries; ++attempt) {

// Генерация списка

DoublyLinkedList<double> list;

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

list.push\_back(generateRandomDouble(-1.0, 1.0));

}

// Тестирование сортировки списка

auto startList = std::chrono::high\_resolution\_clock::now();

list.selectionSort(listPassCount, listSwapCount);

auto endList = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsedList = endList - startList;

listTimes.push\_back(elapsedList.count());

}

}

// Вывод результатов

std::cout << "size: " << size << "\n";

std::cout << "Spisok:\n";

std::cout << " Srednee vremya: " << std::accumulate(listTimes.begin(), listTimes.end(), 0.0) / listTimes.size() << " second \n";

std::cout << " Count prohodov (pass): " << listPassCount << "\n";

std::cout << " Count swap: " << listSwapCount << "\n";

std::cout << "----------------------------------------\n";

}

}

// Тест для проверки операций вставки и удаления

void testStringOperations() {

DoublyLinkedList<std::string> list;

// Исходные данные

std::string strings[] = {"apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"};

// Заполняем список

for (const auto& str : strings) {

list.push\_back(str);

}

// Выводим список до операций

std::cout << "Spisok do operacii:\n";

list.print();

// Вставляем элемент "mango" на позицию 5

list.insert("mango", 5);

std::cout << "\nAfter vstavki 'mango' na position 5:\n";

list.print();

// Удаляем элемент на позиции 2

list.remove(2);

std::cout << "\nAfter delite element on position 2:\n";

list.print();

// Вставляем элемент "orange" в начало

list.push\_front("orange");

std::cout << "\nAfter vstavki 'orange' in start:\n";

list.print();

// Удаляем элемент в конце

list.remove(list.getSize() - 1);

std::cout << "\nAfter delite poslednego delite:\n";

list.print();

}

void testIntegers() {

DoublyLinkedList<int> list;

for (int i = 0; i < 1000; ++i) {

list.push\_back(rand() % 2001 - 1000);

}

int sum = 0, min = 1001, max = -1001;

for (auto& item : list) {

sum += item;

if (item < min) min = item;

if (item > max) max = item;

}

double average = static\_cast<double>(sum) / list.getSize();

std::cout << "Sum: " << sum << "\n";

std::cout << "Average: " << average << "\n";

std::cout << "Min: " << min << "\n";

std::cout << "Max: " << max << "\n";

}

struct Person {

std::string surname;

std::string name;

std::string patronymic;

std::string dob;

};

void testPersons() {

DoublyLinkedList<Person> list;

std::string surnames[] = {"First", "Second", "Tri", "Four", "Five"};

std::string names[] = {"Masha", "Mary", "Katy", "Nastya", "Liza"};

std::string patronymics[] = {"A.", "B.", "C.", "D.", "E."};

for (int i = 0; i < 100; ++i) {

Person p;

p.surname = surnames[rand() % 5];

p.name = names[rand() % 5];

p.patronymic = patronymics[rand() % 5];

p.dob = std::to\_string(1980 + rand() % 40) + "-" + std::to\_string(1 + rand() % 12) + "-" + std::to\_string(1 + rand() % 28);

list.push\_back(p);

}

DoublyLinkedList<Person> young, old;

for (auto& person : list) {

int year = std::stoi(person.dob.substr(0, 4));

if (2023 - year < 20) young.push\_back(person);

if (2023 - year > 30) old.push\_back(person);

}

std::cout << "Young (<20): " << young.getSize() << "\n";

std::cout << "Old (>30): " << old.getSize() << "\n";

}

void testShuffleAndSort() {

DoublyLinkedList<int> list;

for (int i = 0; i < 10; ++i) {

list.push\_back(i);

}

std::cout << "Ishodnyi spisok:\n";

list.print();

list.shuffle();

std::cout << "\nAfter peremeshivaniya:\n";

list.print();

size\_t passCount, swapCount;

list.selectionSort(passCount, swapCount);

std::cout << "\nAfter sort:\n";

list.print();

std::cout << "Count prohodov: " << passCount << "\n";

std::cout << "Count swap: " << swapCount << "\n";

}

int main() {

srand(static\_cast<unsigned>(time(0)));

testStringOperations();

testIntegers();

testPersons();

testShuffleAndSort();

testSortingPerformance();

return 0;

}

**Результат:**

C:\Users\user\CLionProjects\untitled16\cmake-build-debug\untitled16.exe

Spisok do operacii:

apple banana cherry date elderberry fig grape honeydew kiwi lemon

After vstavki 'mango' na position 5:

apple banana cherry date elderberry mango fig grape honeydew kiwi lemon

After delite element on position 2:

apple banana date elderberry mango fig grape honeydew kiwi lemon

After vstavki 'orange' in start:

orange apple banana date elderberry mango fig grape honeydew kiwi lemon

After delite poslednego delite:

orange apple banana date elderberry mango fig grape honeydew kiwi

Sum: 4693

Average: 4.693

Min: -995

Max: 998

Young (<20): 36

Old (>30): 27

Ishodnyi spisok:

0 1 2 3 4 5 6 7 8 9

After peremeshivaniya:

7 6 8 1 9 0 4 2 3 5

Count prohodov: 45

Count swap: 9

size: 1000

Spisok:

Srednee vremya: 0.000776118 second

Count prohodov (pass): 499500

Count swap: 995

----------------------------------------

size: 2000

Spisok:

Srednee vremya: 0.00322 second

Count prohodov (pass): 1999000

Count swap: 1997

----------------------------------------

size: 4000

Spisok:

Srednee vremya: 0.0128332 second

Count prohodov (pass): 7998000

Count swap: 3992

----------------------------------------

size: 8000

Spisok:

Srednee vremya: 0.0511956 second

Count prohodov (pass): 31996000

Count swap: 7990

----------------------------------------

size: 16000

Spisok:

Srednee vremya: 0.223661 second

Count prohodov (pass): 127992000

Count swap: 15989

----------------------------------------

size: 32000

Spisok:

Srednee vremya: 1.11628 second

Count prohodov (pass): 511984000

Count swap: 31986

----------------------------------------

size: 64000

Spisok:

Srednee vremya: 5.16268 second

Count prohodov (pass): 2047968000

Count swap: 63995

----------------------------------------

size: 128000

Spisok:

Srednee vremya: 21.434 second

Count prohodov (pass): 8191936000

Count swap: 127986

----------------------------------------

Process finished with exit code 0

**1. Производительность (время выполнения)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер списка** | **Среднее время (секунды)** | **Размер массива** | **Среднее время (миллисекунды)** |
| 1000 | 0.000776 | 1000 | ~1.75 |
| 2000 | 0.00322 | 2000 | ~8.5 |
| 4000 | 0.012833 | 4000 | ~33 |
| 8000 | 0.051195 | 8000 | ~130 |
| 16000 | 0.223661 | 16000 | ~520 |
| 32000 | 1.11628 | 32000 | ~2100 |
| 64000 | 5.16268 | 64000 | ~8500 |
| 128000 | 21.434 | 128000 | ~34000 |

**Сравнение**

* **Двусвязный список**:
  + Время выполнения растёт квадратично (O(n²)), что соответствует теоретической сложности сортировки выбором.
* **Массив**:
  + Время выполнения также растёт квадратично (O(n²)), но абсолютные значения времени значительно меньше.

**Вывод**

* **Массив работает быстрее**, чем двусвязный список, для всех размеров данных.
* Это связано с тем, что операции с массивом (доступ по индексу, обмен элементов) выполняются быстрее, чем аналогичные операции в двусвязном списке (обход узлов, изменение указателей).

**2. Количество проходов (passes)**

**Двусвязный список**

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер списка** | **Количество проходов** | **Размер массива** | **Количество проходов** |
| 1000 | 499500 | 1000 | 499500 |
| 2000 | 1999000 | 2000 | 1999000 |
| 4000 | 7998000 | 4000 | 7998000 |
| 8000 | 31996000 | 8000 | 31996000 |
| 16000 | 127992000 | 16000 | 127992000 |
| 32000 | 511984000 | 32000 | 511984000 |
| 64000 | 2047968000 | 64000 | 2047968000 |
| 128000 | 8191936000 | 128000 | 8191936000 |

**Сравнение**

* Количество проходов **одинаково** для двусвязного списка и массива.
* Это связано с тем, что алгоритм сортировки выбором всегда выполняет **n(n-1)/2** сравнений, где **n** — количество элементов.

**Вывод**

* Количество проходов не зависит от структуры данных (список или массив).

**3. Количество обменов (swaps)**

**Двусвязный список**

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер списка** | **Количество обменов** | **Размер массива** | **Количество обменов** |
| 1000 | 995 | 1000 | ~992 |
| 2000 | 1997 | 2000 | ~1992 |
| 4000 | 3992 | 4000 | ~3990 |
| 8000 | 7990 | 8000 | ~7990 |
| 16000 | 15989 | 16000 | ~15990 |
| 32000 | 31986 | 32000 | ~31990 |
| 64000 | 63995 | 64000 | ~63990 |
| 128000 | 127986 | 128000 | ~127990 |

**Сравнение**

* Количество обменов **практически одинаково** для двусвязного списка и массива.
* Это связано с тем, что количество обменов зависит от входных данных и алгоритма, а не от структуры данных.

**Вывод**

* Количество обменов не зависит от структуры данных (список или массив).

**Итоговое сравнение**

|  |  |  |  |
| --- | --- | --- | --- |
| **Параметр** | **Двусвязный список** | **Массив** | **Вывод** |
| **Время выполнения** | Медленнее (O(n²)) | Быстрее (O(n²)) | Массив быстрее |
| **Количество проходов** | Одинаково | Одинаково | Нет разницы |
| **Количество обменов** | Одинаково | Одинаково | Нет разницы |

1. **Если важна скорость**, можно использовать массив, так как он работает быстрее благодаря прямому доступу к элементам по индексу.
2. **Если важна гибкость**, можно использовать двусвязный список, если требуется частое добавление/удаление элементов в середине списка.
3. **Для больших данных** (например, 128000 элементов) массив предпочтительнее из-за значительно меньшего времени выполнения.

# Заключение.

В ходе выполнения лабораторной работы была реализована структура данных "двусвязный список" на языке C++.

1. **Применимость двусвязного списка**:
   * Список эффективен для задач, где требуется частое добавление и удаление элементов в начало и конец.
   * Менее эффективен для задач, требующих частого доступа к элементам по индексу.
2. **Сложность реализации**:
   * Реализация двусвязного списка требует внимательной работы с указателями.
   * Сортировка выбором для списка оказалась менее эффективной по сравнению с массивами из-за необходимости обхода элементов.

* Для задач, требующих частого доступа к элементам по индексу, лучше использовать массивы или векторы.