МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных технологий

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 5
по дисциплине
«Структуры и алгоритмы обработки данных**»**

Тема: «Однонаправленный динамический список»

Выполнил студент группы Зуев Д.А.

Принял преподаватель Сартаков М.В.

Самостоятельная работа выполнена «__»_____202__г. *(подпись студента)*

«Зачтено» «__»_____202__г. *(подпись руководителя)*

Москва 2023

Практическая работа 5

1. Цель

Получить знания и практические навыки управления динамическим однонаправленным списком.

2. Постановка задачи

Реализуйте программу решения задачи варианта по использованию линейного однонаправленного списка.

1. Разработать функцию для создания исходного списка, его вывода и добавления узла.
2. Информационная часть узла списка определена вариантом.
3. Разработать функции дополнительного задания варианта.
4. В основной программе выполнить тестирование каждой функции, описанной в задании. Программа должна позволять пользователю непрерывно выполнять операции над списком в произвольном порядке.
5. Составить отчет по выполненному заданию. В отчет включить ответы на вопросы к практической работе.

Вариант 29. Тип информационной части узла - int. Даны два однонаправленных списка L1 и L2. Определить, содержит ли список L1 список L2 как подсписок. Сформировать массив номеров элементов списка L2, которые не содержаться в списке L1. Удалить из списка L2 узлы, номера которых сохранены в массиве.

3. Решение

Наша программа предоставляет интерфейс для работы с односвязными списками L1 и L2. Она позволяет пользователю добавлять элементы в списки, проверять наличие подсписка, находить элементы, которые не содержатся в другом списке, и удалять их. Все основные функции реализованы в виде отдельных блоков кода для обработки узлов списка и выполнения действий с ними.

Структура Node представляет собой узел односвязного списка и содержит два поля: data — для хранения значения узла и next — указатель на следующий узел списка. Эта структура является основой для формирования цепочек данных, объединённых в список.

```
struct Node {
    int data;
    Node* next;
};
```

Функция createNode создает новый узел с заданным значением. Она выделяет память для нового узла, инициализирует его полем data, устанавливает указатель next в nullptr и возвращает указатель на новый узел. Эта функция упрощает создание новых элементов списка.

```
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = nullptr;
    return newNode;
}
```

Функция appendNode добавляет новый узел в конец списка. Она сначала проверяет, пустой ли список, и, если да — делает новый узел его началом. Если список не пуст, функция находит последний узел и присоединяет к нему новый

элемент. Эта функция позволяет последовательно формировать список.

```
void appendNode(Node*& head, int data) {
    Node* newNode = createNode(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

Функция printList выводит значения всех узлов списка на экран, разделяя их пробелами. Она начинает с начала списка и проходит по всем элементам до конца, последовательно выводя значения каждого узла, что позволяет визуально оценить содержимое списка.

```
void printList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        std::cout << temp->data << " ";
        temp = temp->next;
    }
    std::cout << std::endl;
}
```

Функция containsSublist проверяет, является ли список L2 подсписком списка L1. Она последовательно проходит по списку L1 и находит совпадения последовательности элементов, сравнивая их с L2. Если вся последовательность L2 найдена в L1, функция возвращает true; иначе — false.

```
bool containsSublist(Node* L1, Node* L2) {
    if (L2 == nullptr) return true; // Пустой список L2 всегда является подсписком
    if (L1 == nullptr) return false; // Пустой список L1 не может содержать непустой подсписок

    Node* currentL1 = L1;
    while (currentL1 != nullptr) {
        Node* tempL1 = currentL1;
        Node* tempL2 = L2;

        // Проверяем подсписок
```

```
        while (tempL1 != nullptr && tempL2 != nullptr && tempL1->data ==
tempL2->data) {
            tempL1 = tempL1->next;
            tempL2 = tempL2->next;
        }

        if (tempL2 == nullptr) return true; // Если прошли весь список L2,
значит он является подсписком
        currentL1 = currentL1->next;
    }

    return false;
}
```

Функция findNonContainedElements возвращает индексы элементов списка L2, которые не содержатся в L1. Она последовательно проверяет каждый элемент из L2 на наличие в L1. Если элемент отсутствует, его индекс добавляется в вектор nonContained. Это позволяет узнать, какие элементы из L2 уникальны.

```
std::vector<int> findNonContainedElements(Node* L1, Node* L2) {
    std::vector<int> nonContained;
    int index = 0;
    Node* tempL2 = L2;

    while (tempL2 != nullptr) {
        Node* tempL1 = L1;
        bool found = false;

        // Проверяем, содержится ли элемент L2 в L1
        while (tempL1 != nullptr) {
            if (tempL1->data == tempL2->data) {
                found = true;
                break;
            }
            tempL1 = tempL1->next;
        }

        if (!found) {
            nonContained.push_back(index);
        }

        tempL2 = tempL2->next;
        index++;
    }

    return nonContained;
```

```
}
```

Функция deleteNodesByIndices удаляет элементы из списка L2 по указанным индексам. Она проходит по L2, сравнивая текущий индекс с каждым из переданных индексов. Если индекс совпадает, узел удаляется. Эта функция помогает эффективно очистить L2 от ненужных элементов.

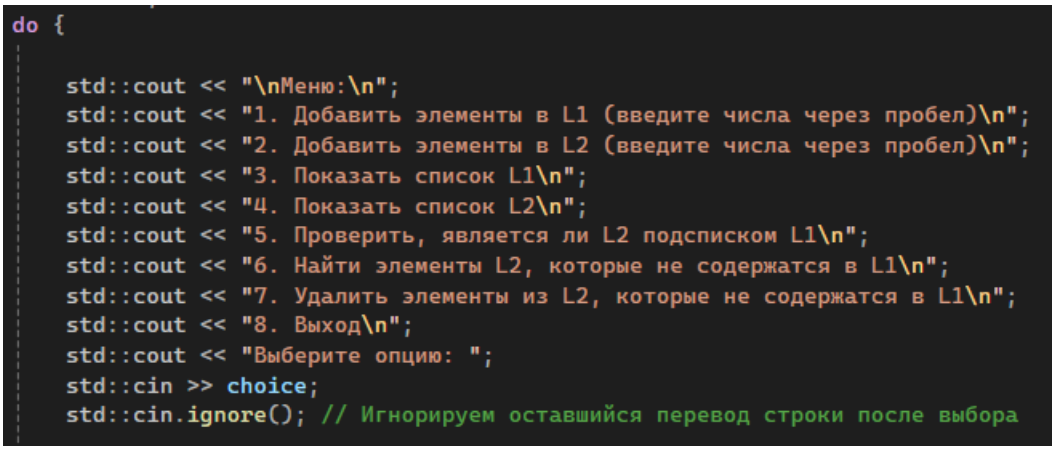
```
void deleteNodesByIndices(Node*& L2, const std::vector<int>& indices) {
    Node* prev = nullptr;
    Node* current = L2;
    int currentIndex = 0;
    int indicesIndex = 0;

    while (current != nullptr && indicesIndex < indices.size()) {
        if (currentIndex == indices[indicesIndex]) {
            if (prev == nullptr) { // Удаление из головы списка
                L2 = current->next;
                delete current;
                current = L2;
            }
            else {
                prev->next = current->next;
                delete current;
                current = prev->next;
            }
            indicesIndex++;
        }
        else {
            prev = current;
            current = current->next;
        }
        currentIndex++;
    }
}
```

Функция appendMultipleNodes добавляет несколько узлов в список, принимая строку чисел, разделённых пробелами. Она использует stringstream для обработки строки, выделяя из неё числа и добавляя их в список. Эта функция упрощает ввод данных, позволяя добавлять несколько узлов за один раз.

```
void appendMultipleNodes(Node*& head, const std::string& input) {
    std::stringstream ss(input);
    int data;
    while (ss >> data) {
```

```
            appendNode(head, data);
        }
    }
```

Функция menu представляет собой интерфейс пользователя и предлагает варианты для работы с двумя списками, L1 и L2. Она предоставляет опции для добавления элементов, вывода списков, проверки подсписка, поиска уникальных элементов и удаления элементов по индексу. Эта функция взаимодействует с пользователем и вызывает нужные функции для выполнения операций.



```
do {

    std::cout << "\nМеню:\n";
    std::cout << "1. Добавить элементы в L1 (введите числа через пробел)\n";
    std::cout << "2. Добавить элементы в L2 (введите числа через пробел)\n";
    std::cout << "3. Показать список L1\n";
    std::cout << "4. Показать список L2\n";
    std::cout << "5. Проверить, является ли L2 подсписком L1\n";
    std::cout << "6. Найти элементы L2, которые не содержатся в L1\n";
    std::cout << "7. Удалить элементы из L2, которые не содержатся в L1\n";
    std::cout << "8. Выход\n";
    std::cout << "Выберите опцию: ";
    std::cin >> choice;
    std::cin.ignore(); // Игнорируем оставшийся перевод строки после выбора
```

Рисунок 1 – Часть функции menu

4. Тестирование

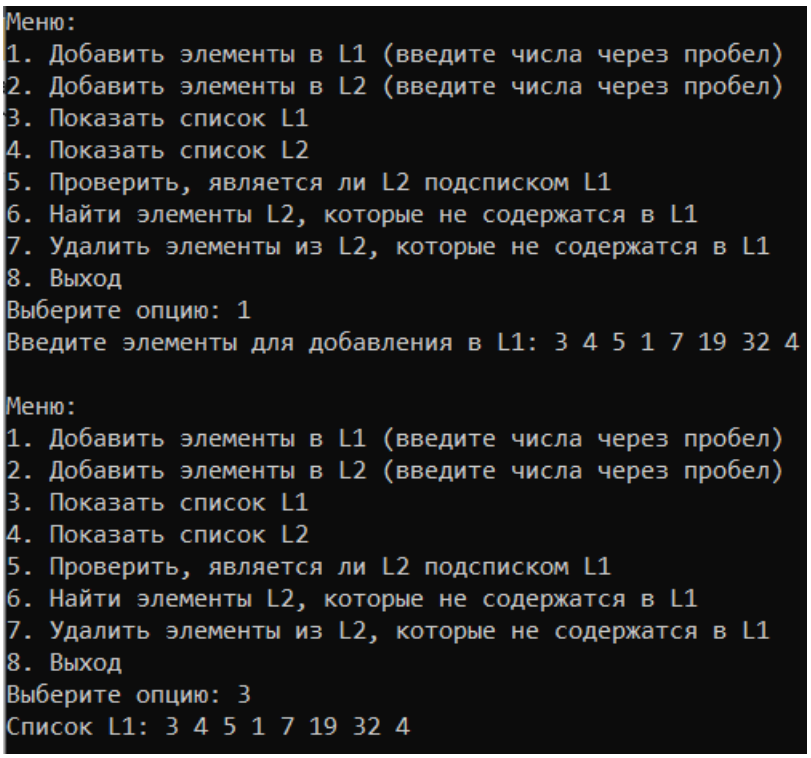


Рисунок 2 – Тестирование добавления элементов в список L1 и вывод его элементов

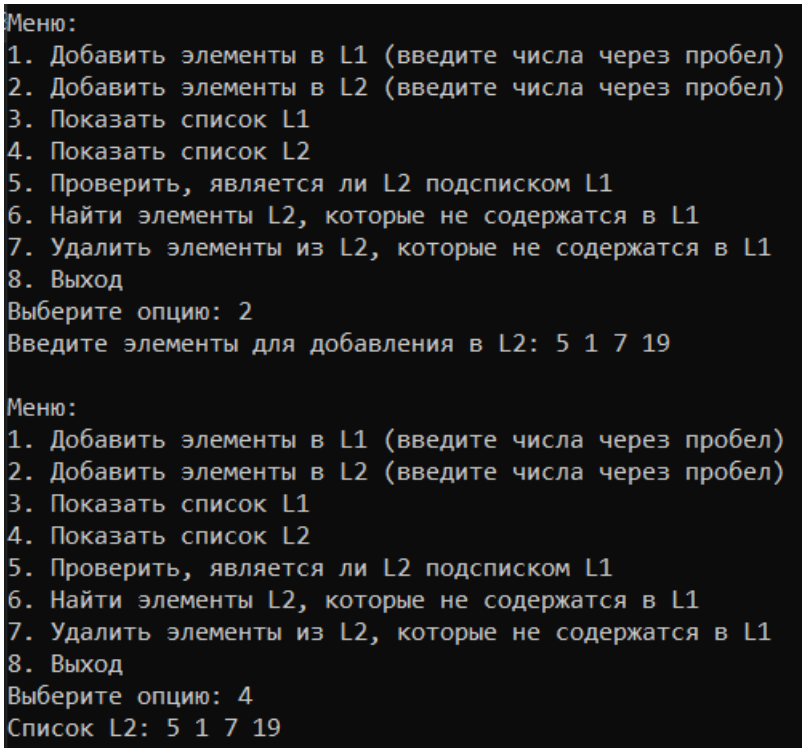


Рисунок 3 – Тестирование добавления элементов в список L2 и вывод его элементов

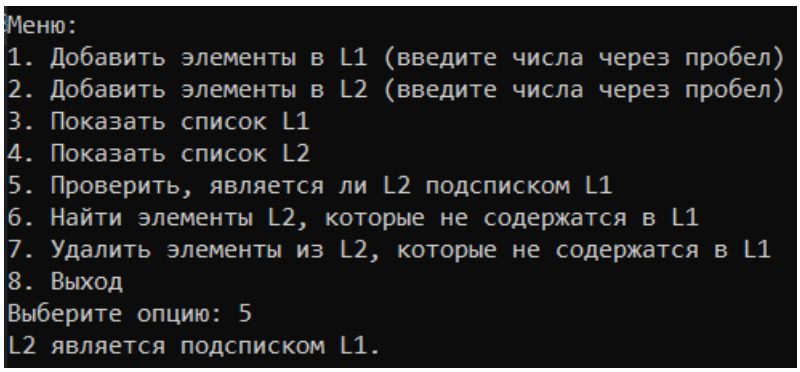


Рисунок 4 – Тестирование проверки является ли L2 подсписком L1

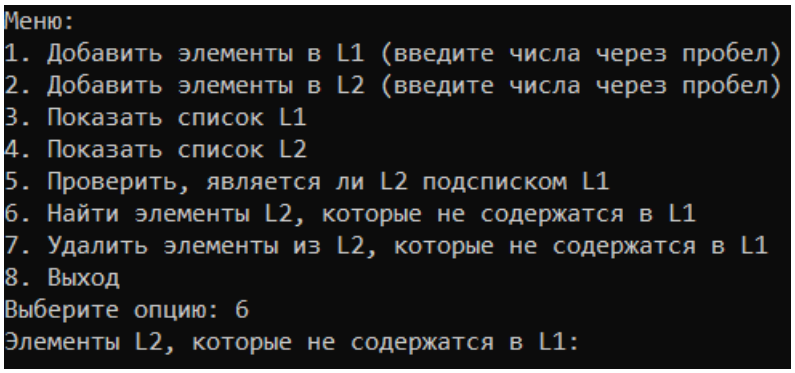


Рисунок 5 – Тестирование нахождения элементов L2, которые не содержатся в L1 (таких нет)

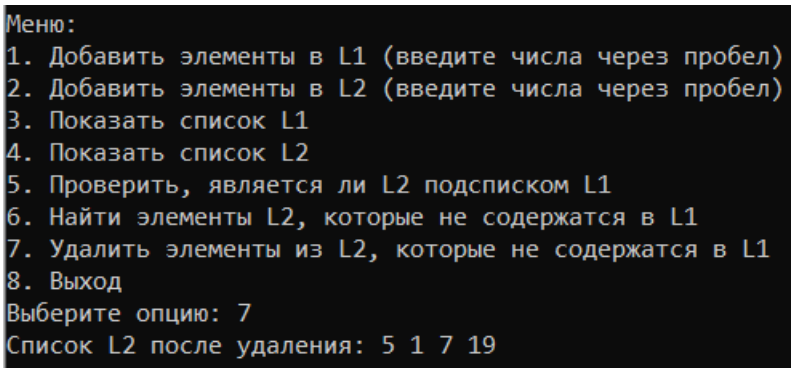


Рисунок 6 – Тестирование удаления элементов из L2, которые не содержатся в L1 (таких нет, список не изменился)

5. Вывод

В программе на C++ реализованы функции для работы с однонаправленными списками: создание узлов, добавление их в конец списка, вывод элементов, проверка подсписка, поиск отсутствующих элементов и удаление узлов по индексам. Пользователь может интерактивно добавлять

элементы через пробелы, проверять, является ли один список подсписком другого, находить и удалять элементы. Программа демонстрирует основные возможности работы с линейными списками и динамическим управлением памятью.

6. Исходный код

```
#include <iostream>
#include <vector>
#include <sstream> // Для использования stringstream

struct Node {
    int data;
    Node* next;
};

// Функция для создания нового узла
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = nullptr;
    return newNode;
}

// Функция для добавления узла в конец списка
void appendNode(Node*& head, int data) {
    Node* newNode = createNode(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Функция для вывода списка
void printList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        std::cout << temp->data << " ";
        temp = temp->next;
    }
```

```
        std::cout << std::endl;
    }

    // Функция для проверки, содержит ли список L1 подсписок L2
    bool containsSublist(Node* L1, Node* L2) {
        if (L2 == nullptr) return true; // Пустой список L2 всегда является
подсписком
        if (L1 == nullptr) return false; // Пустой список L1 не может
содержать непустой подсписок

        Node* currentL1 = L1;
        while (currentL1 != nullptr) {
            Node* tempL1 = currentL1;
            Node* tempL2 = L2;

            // Проверяем подсписок
            while (tempL1 != nullptr && tempL2 != nullptr && tempL1->data ==
tempL2->data) {
                tempL1 = tempL1->next;
                tempL2 = tempL2->next;
            }

            if (tempL2 == nullptr) return true; // Если прошли весь список
L2, значит он является подсписком
            currentL1 = currentL1->next;
        }

        return false;
    }

    // Функция для формирования массива номеров элементов L2, которые не
содержатся в L1
    std::vector<int> findNonContainedElements(Node* L1, Node* L2) {
        std::vector<int> nonContained;
        int index = 0;
        Node* tempL2 = L2;

        while (tempL2 != nullptr) {
            Node* tempL1 = L1;
            bool found = false;

            // Проверяем, содержится ли элемент L2 в L1
            while (tempL1 != nullptr) {
                if (tempL1->data == tempL2->data) {
                    found = true;
                    break;
                }
                tempL1 = tempL1->next;
            }
```

```
        if (!found) {
            nonContained.push_back(index);
        }

        tempL2 = tempL2->next;
        index++;
    }

    return nonContained;
}

// Функция для удаления узлов из L2 по номерам
void deleteNodesByIndices(Node*& L2, const std::vector<int>& indices) {
    Node* prev = nullptr;
    Node* current = L2;
    int currentIndex = 0;
    int indicesIndex = 0;

    while (current != nullptr && indicesIndex < indices.size()) {
        if (currentIndex == indices[indicesIndex]) {
            if (prev == nullptr) { // Удаление из головы списка
                L2 = current->next;
                delete current;
                current = L2;
            }
            else {
                prev->next = current->next;
                delete current;
                current = prev->next;
            }
            indicesIndex++;
        }
        else {
            prev = current;
            current = current->next;
        }
        currentIndex++;
    }
}

// Функция для добавления нескольких элементов в список
void appendMultipleNodes(Node*& head, const std::string& input) {
    std::stringstream ss(input);
    int data;
    while (ss >> data) {
        appendNode(head, data);
    }
}
```

```
// Меню для работы со списком
void menu() {
    setlocale(0, "");

    Node* L1 = nullptr;
    Node* L2 = nullptr;
    int choice;
    do {

        std::cout << "\nМеню:\n";
        std::cout << "1. Добавить элементы в L1 (введите числа через
пробел)\n";
        std::cout << "2. Добавить элементы в L2 (введите числа через
пробел)\n";
        std::cout << "3. Показать список L1\n";
        std::cout << "4. Показать список L2\n";
        std::cout << "5. Проверить, является ли L2 подсписком L1\n";
        std::cout << "6. Найти элементы L2, которые не содержатся в
L1\n";
        std::cout << "7. Удалить элементы из L2, которые не содержатся в
L1\n";
        std::cout << "8. Выход\n";
        std::cout << "Выберите опцию: ";
        std::cin >> choice;
        std::cin.ignore(); // Игнорируем оставшийся перевод строки после
выбора

        switch (choice) {
        case 1: {
            std::string input;
            std::cout << "Введите элементы для добавления в L1: ";
            std::getline(std::cin, input);
            appendMultipleNodes(L1, input);
            break;
        }
        case 2: {
            std::string input;
            std::cout << "Введите элементы для добавления в L2: ";
            std::getline(std::cin, input);
            appendMultipleNodes(L2, input);
            break;
        }
        case 3:
            std::cout << "Список L1: ";
            printList(L1);
            break;
        case 4:
            std::cout << "Список L2: ";
            printList(L2);
            break;
```

```
            case 5:
                if (containsSublist(L1, L2)) {
                    std::cout << "L2 является подсписком L1." << std::endl;
                }
                else {
                    std::cout << "L2 не является подсписком L1." <<
std::endl;
                }
                break;
            case 6: {
                std::vector<int> nonContained = findNonContainedElements(L1,
L2);
                std::cout << "Элементы L2, которые не содержатся в L1: ";
                for (int index : nonContained) {
                    std::cout << index << " ";
                }
                std::cout << std::endl;
                break;
            }
            case 7: {
                std::vector<int> nonContained = findNonContainedElements(L1,
L2);
                deleteNodesByIndices(L2, nonContained);
                std::cout << "Список L2 после удаления: ";
                printList(L2);
                break;
            }
            case 8:
                std::cout << "Выход из программы." << std::endl;
                break;
            default:
                std::cout << "Неверный выбор. Пожалуйста, попробуйте еще
раз." << std::endl;
            }
        } while (choice != 8);
    }

    // Тестирование
    int main() {
        menu();
        return 0;
    }
```

7. Ответы на вопросы после практической работы

1. Три уровня представления данных в программной системе:

- Физический: как данные хранятся в памяти (биты и байты).

- Логический: типы данных и их структуры (массивы, списки).
- Концептуальный: абстракция данных, то, как данные представляются пользователю (таблицы, графы).

2. Тип данных определяет:

- Формат, операции, и возможные значения, которые может принимать переменная.

3. Структура данных определяет:

- Способ организации данных и связи между ними (например, массив, стек, дерево).

4. Структуры хранения данных в компьютерных технологиях:

- Данные могут храниться в памяти компьютера, на жестких дисках, в кэше и других носителях. К основным структурам хранения относят массивы, списки, деревья и графы.

5. Линейная структура данных:

- Данные расположены последовательно, каждый элемент имеет точно одного предшественника и одного последователя, кроме первого и последнего элементов.

6. Структура данных "линейный список":

- Последовательная структура данных, в которой каждый элемент связан с предыдущим и/или следующим элементом (например, связный список).

7. Стек:

- Линейная структура данных с доступом к элементам по принципу LIFO (последний вошел — первый вышел).

8. Очередь:

- Линейная структура данных с доступом по принципу FIFO (первый вошел — первый вышел).

9. Отличие стека от линейного списка:

- В стеке элементы доступны только с одного конца (вход и выход), в линейном списке можно произвольно вставлять и удалять элементы.

10. Какой линейный список лучше использовать для вывода последовательности в обратном порядке:

- Двусвязный список или стек.

11. Сложность вставки элемента в i-ую позицию:

- а) Массив: O(n) из-за необходимости сдвига элементов.
- б) Линейный список: O(i) для нахождения позиции, плюс O(1) для вставки.

12. Сложность удаления элемента из i-ой позиции:

- а) Массив: O(n) из-за сдвига.
- б) Линейный список: O(i) для поиска позиции, плюс O для удаления.

13. Трюк Вирта при удалении элемента из списка:

- Использование "призрачного" узла для упрощения манипуляций с элементами в списке, что уменьшает количество условий в коде.

14. Структура узла однонаправленного списка:

- Содержит данные и указатель на следующий узел.

15. Алгоритм вывода линейного однонаправленного списка:

```
void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->info << " ";
        current = current->next;
    }
}
```

16. Перемещение последнего элемента в начало списка (C++ код):

```
void moveLastToFront(Node*& head) {
    if (!head || !head->next) return;
    Node* secondLast = nullptr;
    Node* last = head;

    while (last->next) {
        secondLast = last;
        last = last->next;
    }
    secondLast->next = nullptr;
    last->next = head;
    head = last;
}
```

17. Лишнее действие в следующем фрагменте кода:

- List L = new List; — это неправильное создание нового списка, необходимо new Node.

- Куда вставляется новый узел: в начало списка после фиктивного узла LL.