# Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

# (СибГУТИ)

09.03.01 Информатика и вычислительная техника

Профиль: Электронно-вычислительные машины, комплексы, системы и сети

(очная форма обучения)

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

# в/на кафедре вычислительных систем СибГУТИ

(наименование профильной организации/структурного подразделения СибГУТИ)

# «РЕАЛИЗАЦИЯ СТЕКА НА БАЗЕ ОДНОСВЯЗНОГО СПИСКА»

Выполнил: Киреев И. А. студент института ИВТ

# гр. ИВ-121 / /

«27» мая 2023 г. (подпись)

# Проверил:

Руководитель от СибГУТИ / /

«27» мая 2023 г. (подпись)

# Новосибирск 2023

**План-график проведения** учебной **практики**

Вид практики

# Киреев Илья Анатольевич

Фамилия Имя Отчество студента

# института Информатика и вычислительная техника , 2 курса, гр. ИВ-121

Направление: 09.03.01 — Информатика и вычислительная техника

Код – Наименование направления (специальности)

# Профиль: Электронно-вычислительные машины, комплексы, системы и сети

Место прохождения практики

Объем практики: **108/3** часов/ЗЕ Вид практики ***учебная***

Тип практики ***ознакомительная***

# Срок практики с "30" января 2023 г.

по "27" мая 2023 г.

# Содержание практики\*:

|  |  |
| --- | --- |
| Наименование видов деятельности | Дата (начало – окончание) |
| 1. Общее ознакомление со структурным подразделением предприятия, вводный инструктаж по технике безопасности | 30.01.2023–01.02.2023 |
| 2. Выдача задания на практику, деление студентов на группы (если необходимо), определение конкретной  индивидуальной темы, формирование плана работ | 02.02.2023–04.02.2023 |
| 3. Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики | 06.02.2023–11.02.2023 |
| 1. Выполнение работ в соответствии с составленным   планом:   * 1. Разработка заголовочного файла с прототипами методов работы со структурой данных   2. Разработка системы сборки проекта СMake   3. Разработка структуры данных в отдельном файле   4. Разработка программы для тестирования полученной библиотеки для работы со структурой данных   5. Отладка и форматирование кода программы | 13.02.2023 – 20.05.2023 |
| 5. Анализ полученных результатов и произведенной работы. Составление отчета по практике, защита отчета | 22.05.2023–27.05.2023 |

\*В соответствии с программой практики

# Руководитель от СибГУТИ / /

«28» 01 2023г. (подпись)

### ЗАДАНИЕ НА ПРАКТИКУ

Реализовать программно, исследовать эффективность и описать структуру данных “Стек” на основе структуры данных «Односвязный список».

### ВВЕДЕНИЕ

Стек - это структура данных, основанная на односвязном списке, которая позволяет хранить и управлять элементами в порядке "последним пришел - первым вышел" (LIFO - Last In, First Out). Односвязный список состоит из узлов, каждый из которых содержит данные и ссылку на следующий узел. Стек на основе односвязного списка предлагает удобные операции добавления и удаления элементов. При добавлении нового элемента в стек, он становится вершиной списка, а его ссылка указывает на предыдущий элемент, образуя цепочку связанных узлов. Извлечение элемента из стека происходит путем изменения ссылки на вершину, что делает предыдущий элемент новой вершиной стека. Таким образом, только последний добавленный элемент может быть извлечен из стека.

### СТРУКТУРА ОДНОСВЯЗНОГО СПИСКА

Односвязный список является важной структурой данных, применяемой в программировании и информатике. Он представляет собой набор узлов, где каждый узел содержит данные и ссылку на следующий узел в списке. Односвязный список отличается от двунаправленного списка тем, что каждый узел содержит ссылку только на следующий узел, а не на предыдущий (Рис. 1).

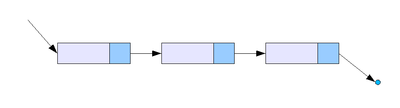


Рис. 1 — Абстрактное представление односвязного списка

Каждый узел односвязного списка состоит из двух основных частей: поля данных, которое хранит сами данные, и поля указателя, которое содержит ссылку на следующий узел в списке. Последний узел списка содержит ссылку на NULL или нулевой указатель, указывающий на конец списка. Односвязный список можно представить в виде последовательности узлов, связанных между собой ссылками. Начальный узел списка называется головой, а конечный узел - хвостом. Структура односвязного списка обеспечивает эффективность и гибкость при добавлении и удалении элементов (Рис. 2).



Рис. 2 — Программное представления односвязного списка

Добавление нового узла в односвязный список осуществляется путем создания нового узла, заполнения его данных и установки ссылки на него в поле указателя предыдущего узла. Таким образом, новый узел становится следующим за предыдущим узлом, образуя связь в списке. Удаление узла из односвязного списка происходит путем изменения ссылки в поле указателя предыдущего узла. Таким образом, предыдущий узел "пропускает" удаленный узел и связывается с последующим узлом, исключая удаленный узел из списка.

Односвязный список позволяет эффективно управлять данными, особенно при операциях вставки и удаления в середине списка. Однако доступ к элементам списка осуществляется последовательно, начиная с головы, что может потребовать больше времени для поиска определенного элемента.

Использование односвязного списка находит применение во многих областях, включая реализацию стеков, очередей, алгоритмов поиска и сортировки, а также связанных структур данных, таких как графы и деревья. Односвязный список обладает гибкостью и эффективностью, что делает его важным инструментом при разработке программного обеспечения.

### СТРУКТУРА СТЕКА

Стек является важной структурой данных, которая работает по принципу "последним пришел - первым вышел" (LIFO - Last In, First Out). Он позволяет хранить и управлять элементами в порядке их добавления, обеспечивая быстрый доступ к последнему добавленному элементу (Рис. 3).

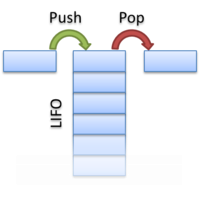


Рис. 3 — принцип работы структуры данных «Стек»

Структура стека может быть реализована на основе различных структур данных, включая односвязный список. Стек на основе односвязного списка представляет собой набор узлов, связанных между собой последовательно. Каждый узел содержит данные и ссылку на следующий узел в стеке. Последний добавленный узел становится вершиной стека, а его ссылка указывает на предыдущую вершину.

Операция добавления элемента в стек, называемая помещением или вставкой, осуществляется путем создания нового узла и установки ссылки на него в вершине стека. Новый узел становится новой вершиной стека, а его ссылка указывает на предыдущую вершину, образуя цепочку связанных узлов.

Операция удаления элемента из стека, называемая извлечением или удалением, происходит путем изменения ссылки в вершине стека на предыдущий узел. Таким образом, предыдущий узел становится новой вершиной стека, и его данные становятся доступными для использования (Рис. 4).

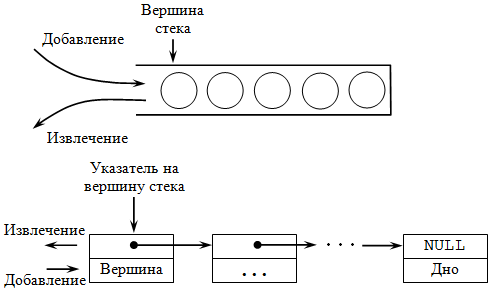


Рис. 4 — Представление операция добавления и извлечения элементов из стека

Структура стека на основе односвязного списка обладает несколькими преимуществами. Во-первых, она позволяет эффективно управлять данными, сохраняя порядок их добавления. Во-вторых, стек на основе односвязного списка может быть реализован динамически, то есть его размер не ограничен и может увеличиваться или уменьшаться по мере необходимости. Стеки на основе односвязного списка широко применяются в различных областях программирования и алгоритмических задачах. Они могут использоваться для реализации алгоритмов обхода деревьев, выполнения обратной польской записи, обработки вызовов функций и многих других задач. Стеки также являются важной составляющей в разработке компиляторов и интерпретаторов.

### АЛГОРИТМ РАБОТЫ

**Добавление элемента**

Добавление элемента в стек на основе связного списка является одной из основных операций, которую необходимо реализовать для работы со стеком. Эта операция позволяет добавить новый элемент на вершину стека.

Процесс добавления элемента в стек на основе связного списка включает несколько шагов. Вначале создается новый узел, который будет содержать добавляемый элемент. Затем устанавливается ссылка этого нового узла на текущую вершину стека. Далее обновляется ссылка вершины стека на новый узел, чтобы он стал новой вершиной.

Сложность операции добавления элемента в стек на основе связного списка составляет O(1), то есть она имеет постоянную временную сложность. При добавлении элемента, независимо от размера стека, требуется выполнить фиксированное количество операций: создание нового узла, обновление ссылок и обновление ссылки на вершину стека. Таким образом, время выполнения операции добавления элемента не зависит от количества элементов в стеке.

Ниже приведен псевдокод для операции добавления элемента в стек на основе связного списка:

procedure push(element):

// Шаг 1: Создание нового узла

new\_node <- create\_node(element)

// Шаг 2: Установка данных нового узла

set\_data(new\_node, element)

// Шаг 3: Установка связи со старым верхним элементом

set\_next(new\_node, top\_of\_stack)

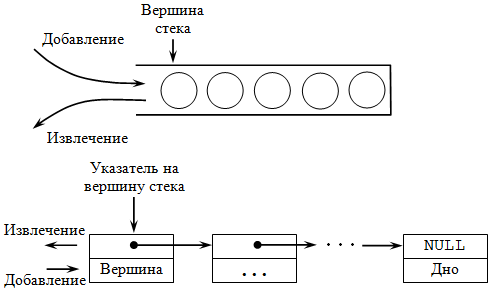
// Шаг 4: Обновление верхнего элемента стека

top\_of\_stack <- new\_node

end procedure

В этом псевдокоде используются следующие ключевые слова и функции:

* create\_node(element): функция, которая создает новый узел и возвращает его
* set\_data(node, element): функция, которая устанавливает данные узла равными указанному элементу
* set\_next(node, next\_node): функция, которая устанавливает ссылку на следующий узел у указанного узла
* top\_of\_stack: переменная, которая хранит ссылку на верхний элемент стека

Рис. 5 — Описание операции добавления элемента в стек на основе связного спика

**Извлечение элемента**

Операция извлечения элемента из стека, также известная как "pop" (выталкивание), является

фундаментальной операцией в структуре данных стек. Стек представляет собой абстрактную структуру данных, которая обладает свойством LIFO (Last-In-First-Out), что означает, что последний элемент, помещенный в стек, будет первым, который будет удален. Извлечение элемента из стека осуществляется путем удаления элемента, находящегося на вершине стека. При этом происходит изменение указателя вершины стека, чтобы он указывал на предыдущий элемент. Таким образом, при извлечении элемента, предыдущий элемент становится новой вершиной стека.

Операция извлечения элемента из стека может быть описана в несколько шагов:

1. Проверка стека на пустоту. Если стек пуст, то операция извлечения невозможна, и может быть возвращено сообщение об ошибке.
2. Если стек не пуст, то текущий элемент, находящийся на вершине стека, считается выбранным для извлечения.
3. Удаление выбранного элемента из стека, что может потребовать изменения указателя вершины стека, чтобы он указывал на предыдущий элемент.
4. Возвращение извлеченного элемента в качестве результата операции извлечения.

Операция извлечения элемента из стека может быть реализована с использованием различных языков программирования и методов, таких как удаление элемента из списка (если стек реализован на основе списка), изменение указателя вершины стека или использование других подходящих методов в зависимости от выбранной реализации.

Таким образом, операция извлечения элемента из стека играет важную роль в обеспечении правильной работы стека, позволяя получать доступ к элементам в порядке, обратном их добавлению (Рис. 5).

Так как «Стек» является достаточно тривиальной структурой данных, в нём отсутствуют операции поиска, удаления конкретного элемента. «Стек» ограничивается лишь несколькими операциями.

### ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ

**Стек**

Исследование эффективности операций стека на основе связного списка является важной задачей, которая позволяет оценить производительность данной структуры данных и оптимизировать ее использование. Операции стека, такие как добавление элемента (push) и удаление элемента (pop), имеют свои сложности, которые должны быть учтены при проектировании и реализации стека на основе связного списка.

Сложность операций стека на основе связного списка зависит от нескольких факторов. Одним из ключевых факторов является время выполнения операций. Время выполнения операции push при использовании связного списка составляет O(1), так как новый элемент добавляется в начало списка, и нет необходимости перестраивать всю структуру данных. Однако, для удаления элемента из связного списка при операции pop необходимо переустановить указатель на следующий элемент, что также требует константного времени O(1).

Также следует учесть сложность по памяти при использовании связного списка для реализации стека. Память, необходимая для хранения элементов, зависит от количества элементов в стеке. Каждый элемент связного списка требует дополнительной памяти для хранения самих данных и указателя на следующий элемент. Поэтому общая сложность по памяти для стека на основе связного списка составляет O(n), где n - количество элементов в стеке. Это связано с тем, что каждый элемент требует дополнительного пространства для указателя на следующий элемент.

Однако, при использовании стека на основе связного списка есть некоторые дополнительные аспекты, которые могут повлиять на эффективность операций. Например, при выполнении операции pop необходимо проверять, не пуст ли стек, чтобы избежать ошибок. Это требует дополнительного времени и проверки условия, что может повлиять на общую производительность.

Также следует учесть, что при использовании стека на основе связного списка возможно возникновение проблемы фрагментации памяти. Поскольку элементы стека хранятся в разных областях памяти и соединяются указателями, может возникнуть ситуация, когда свободные фрагменты памяти разбросаны по всему адресному пространству. Это может снизить эффективность работы стека и требовать дополнительных усилий.

**Односвязный список.**

Исследование эффективности операций связного списка является важной задачей при проектировании и использовании этой структуры данных. Связный список представляет собой динамическую структуру данных, которая состоит из узлов, каждый из которых содержит данные и ссылку на следующий узел. При исследовании эффективности операций связного списка необходимо учитывать их сложности как по времени, так и по памяти.

Одной из основных операций со связным списком является добавление нового элемента в список. При операции добавления элемента в начало списка, так называемая операция prepend, время выполнения является константным O(1), поскольку требуется только создание нового узла и перенаправление ссылки на следующий узел. Однако, при добавлении элемента в конец списка, необходимо пройти через все узлы до последнего, чтобы установить ссылку на новый узел, что требует времени, пропорционального размеру списка. Следовательно, время выполнения операции append в связном списке составляет O(n), где n - количество элементов в списке.

Еще одной важной операцией является удаление элемента из связного списка. При удалении элемента из начала списка, время выполнения также является константным O(1), поскольку требуется только перенаправление ссылки на следующий узел. Однако, для удаления элемента из конца списка, необходимо пройти через все узлы до предпоследнего и перенаправить его ссылку на NULL, что также требует времени, пропорционального размеру списка. Следовательно, время выполнения операции удаления из конца списка в связном списке также составляет O(n) (Рис. 6).

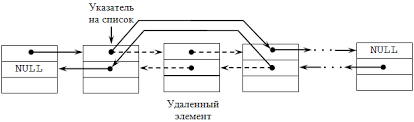


Рис. 6 — Операция удаления элемента из связного списка

Сложность операций связного списка также зависит от использования индексов. Доступ к элементу по индексу в связном списке не является эффективной операцией, так как необходимо последовательно проходить через узлы до требуемого индекса. В худшем случае, когда требуется доступ к последнему элементу списка, время выполнения операции доступа по индексу составляет O(n).

### ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ОДНОСВЯЗНОГО СПИСКА

Программная реализация односвязного списка представлена в данном коде. Он включает в себя определение шаблонного класса list, который представляет собой односвязный список элементов типа T. Класс list реализует основные операции работы со списком, такие как добавление и удаление элементов, поиск элементов, получение размера списка и другие операции.

В коде используется шаблонный класс node, который представляет узел списка и содержит значение элемента и указатель на следующий узел списка.

Методы шаблонного класса list включают:

* Конструкторы и деструктор: конструктор по умолчанию, конструктор с использованием std::initializer\_list, конструктор копирования, конструктор перемещения и деструктор.
* Функции-члены для проверки списка на пустоту и получения его размера.
* Функции-члены для добавления и удаления элементов в начале и конце списка.
* Функции-члены для удаления элементов по индексу, первому вхождению и последнему вхождению указанного значения.
* Функция-член для удаления последовательных повторяющихся элементов в списке.
* Функции-члены для проверки наличия элемента в списке и подсчета количества элементов с указанным значением.
* Операторы индексации для доступа к элементам списка.
* Функции-члены для очистки списка, копирования содержимого другого списка и слияния списка с другим списком.

Методы шаблонного класса list реализованы в соответствии с принципами объектно-ориентированного программирования и обеспечивают эффективную работу со списком.

Также, было проведено тестирование данного списка. Для этого была использована библиотека Gtest. Ниже описаны основные процессы, которые тестируются в каждом тестовом случае:

1. TEST(List, Containing): Тест проверяет функцию contains, которая должна возвращать true, если элемент присутствует в списке, и false в противном случае. В тесте проверяются различные случаи, включая наличие и отсутствие элементов в списке.
2. TEST(List, Size): Тест проверяет функцию size, которая должна возвращать количество элементов в списке. В тесте проверяются различные случаи, включая пустой список и списки с несколькими элементами.
3. TEST(List, BeginEnd): Тест проверяет функции begin и end, которые возвращают итераторы на первый и последний элементы списка соответственно. Тест проверяет, что значения элементов, на которые указывают итераторы, соответствуют ожидаемым значениям.
4. TEST(List, Count): Тест проверяет функцию count, которая должна возвращать количество вхождений заданного элемента в список. В тесте создается список с несколькими повторяющимися элементами, и проверяется, что функция count возвращает ожидаемые значения.
5. TEST(List, IdxOperator): Тест проверяет оператор индексирования [], который позволяет получить доступ к элементу списка по индексу. В тесте проверяется, что значения элементов, полученных с помощью оператора [], соответствуют ожидаемым значениям.
6. TEST(List, Addition): Тест проверяет операции добавления элементов в список. В тесте используются функции push\_back и push\_front для добавления элементов в конец и начало списка соответственно. Затем проверяется, что полученный список совпадает с ожидаемым списком.
7. TEST(List, Removal): Тест проверяет операции удаления элементов из списка. В тесте используются функции pop\_back, pop\_front, erase\_first, erase\_last и erase для удаления элементов из списка. Затем проверяется, что полученный список совпадает с ожидаемым списком.
8. TEST(List, Unique): Тест проверяет функцию unique, которая должна удалить повторяющиеся элементы из списка, оставив только уникальные. В тесте создается список с повторяющимися элементами, и после применения функции unique проверяется, что полученный список совпадает с ожидаемым списком.
9. TEST(List, Merger): Тест проверяет функцию merge, которая должна объединить два списка в один, сохраняя порядок элементов. В тесте создаются два списка и применяется функция merge. Затем проверяется, что полученный список совпадает с ожидаемым списком.
10. TEST(List, RO5): Тест проверяет операции копирования и перемещения для списка. В тесте создаются списки и применяются операции копирования и перемещения. Затем проверяется, что полученные списки совпадают с ожидаемыми списками.

Каждый тестовый случай использует утверждения ASSERT\_EQ, которые сравнивают фактические значения с ожидаемыми значениями и выводят сообщение об ошибке, если значения не совпадают. В результате выполнения тестов был получен результат, представленный на рисунке 7.

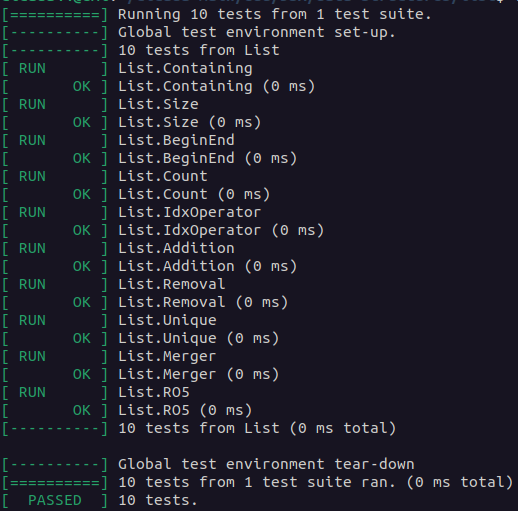


Рис. 7 — Запуск тестов для связного списка

### ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СТЕКА

### Стек основан, а конкретнее, унаследован от структуры данных «Список». Описание реализации представлено ниже.

### Класс stack является наследником класса list, что позволяет использовать функциональность списка для реализации стека.

1. В классе stack определены конструкторы:
   * Конструктор по умолчанию stack(), который создает пустой стек.
   * Конструктор с использованием списка инициализации stack(const std::initializer\_list<T>& init\_list), который создает стек и инициализирует его значениями из переданного списка инициализации.
2. В классе stack определены функции-члены:
   * size\_t size() const: Возвращает количество элементов в стеке, используя функцию size из базового класса list.
   * bool is\_empty() const: Проверяет, является ли стек пустым, используя функцию is\_empty из базового класса list.
   * node<T>\* begin() const: Возвращает указатель на первый элемент в стеке, используя функцию begin из базового класса list.
   * node<T>\* end() const: Возвращает указатель на последний элемент в стеке, используя функцию end из базового класса list.
   * const node<T>\* cbegin() const: Возвращает константный указатель на первый элемент в стеке, используя функцию cbegin из базового класса list.
   * const node<T>\* cend() const: Возвращает константный указатель на последний элемент в стеке, используя функцию cend из базового класса list.
   * void push(const T& value): Добавляет элемент value в верхушку стека, используя функцию push\_front из базового класса list.
   * void pop(): Удаляет элемент из верхушки стека, используя функцию pop\_front из базового класса list.
   * bool contains(const T& value) const: Проверяет, содержится ли элемент value в стеке, используя функцию contains из базового класса list.
   * size\_t count(const T& v) const: Возвращает количество элементов со значением v в стеке, используя функцию count из базового класса list.
   * void print\_stack(std::ostream& os) const: Вспомогательная функция для тестирования, которая выводит содержимое стека в поток os.
3. Перегружен оператор == для сравнения двух стеков. Он использует перегруженный оператор == из базового класса list.
4. Реализованы операторы присваивания для копирования и перемещения:
   * Копирующий оператор присваивания stack& operator=(const stack<T>& s), который очищает текущий стек и копирует содержимое стека s в текущий стек с помощью функции copy из базового класса list.
   * Оператор присваивания перемещения stack& operator=(stack<T>&& s), который просто делегирует операцию присваивания с помощью operator=.

Таким образом, класс stack предоставляет интерфейс для работы со стеком, используя функциональность списка.

### Также было проведено тестирование данной реализации стека:

1. Используется фреймворк Google Test (gtest) для написания и запуска модульных тестов. Функция main инициализирует gtest и запускает все тесты с помощью RUN\_ALL\_TESTS().
2. Определены несколько тестовых случаев с помощью макроса TEST, который принимает имя теста и код тестового случая.
3. В тестовом случае Stack, Containing проверяется функциональность метода contains класса stack. Проверяется, содержится ли определенное значение в стеке. Используются утверждения ASSERT\_EQ, чтобы сравнить ожидаемый результат с фактическим.
4. В тестовом случае Stack, Size проверяется функциональность метода size класса stack. Проверяется, возвращает ли метод правильное количество элементов в стеке.
5. В тестовом случае Stack, BeginEnd проверяется функциональность методов begin, end, cbegin и cend класса stack. Проверяется, возвращают ли эти методы указатели на правильные элементы в стеке.
6. В тестовом случае Stack, Count проверяется функциональность метода count класса stack. Проверяется, возвращает ли метод правильное количество элементов со значением v в стеке.
7. В тестовом случае Stack, Addition проверяется функциональность метода push класса stack. Создается пустой стек, и в него последовательно добавляются элементы с помощью метода push. Проверяется, что стек содержит ожидаемые элементы после добавления.
8. В тестовом случае Stack, Removal проверяется функциональность метода pop класса stack. Создается стек с некоторыми элементами, и вызывается метод pop для удаления элемента из стека. Проверяется, что стек содержит ожидаемые элементы после удаления.
9. В тестовом случае Stack, RO5 проверяется функциональность операторов присваивания и перемещения класса stack. Проверяется, что операторы копирования и перемещения правильно работают и создают стеки с ожидаемыми значениями.
10. В функции main инициализируется gtest и запускаются все тесты с помощью RUN\_ALL\_TESTS().

### В результате запуска тестов, был получен результат, представленный на рисунке 8.

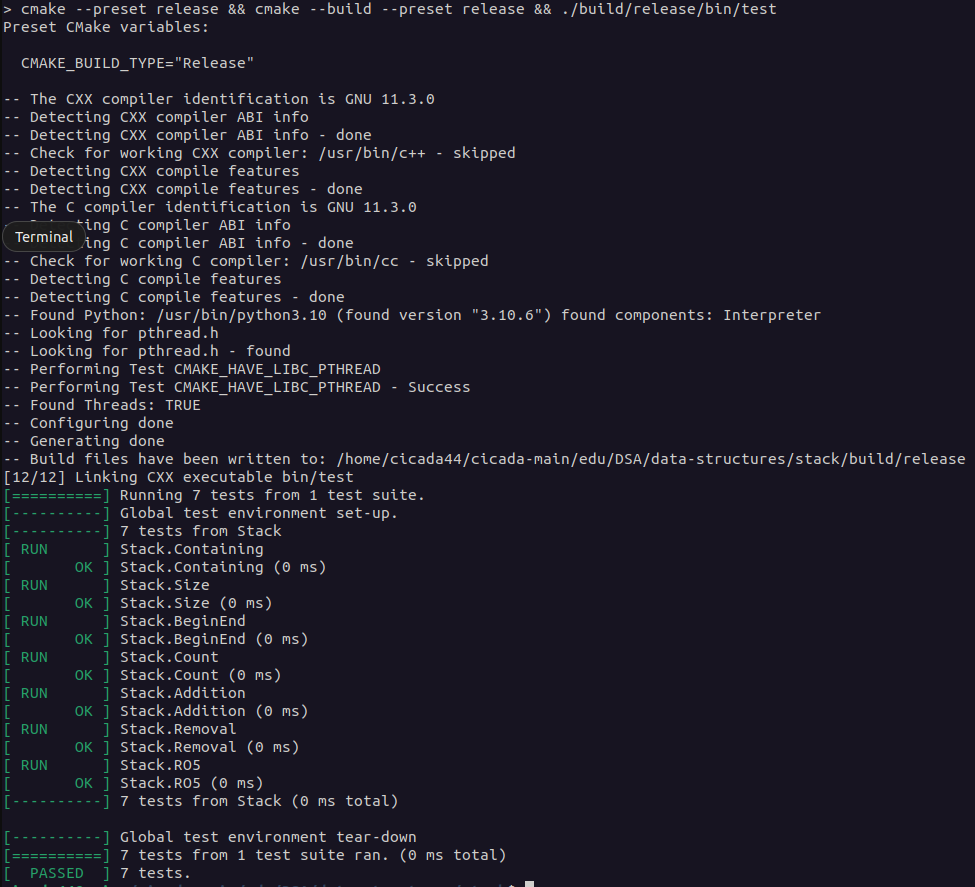
### 

### 

### Рис. 8 — Результат тестирования стека

### РЕЗУЛЬТАТЫ РЕАЛИЗАЦИИ

Компиляция проекта и запуск тестов:



### ЗАКЛЮЧЕНИЕ

В ходе работы была изучена и реализована структура данных "стек" на основе связного списка. Был создан класс stack, предоставляющий функциональность работы со стеком. Основные свойства реализации стека на основе связного списка:

1. Стек представляет собой коллекцию элементов, где добавление новых элементов и удаление существующих происходит только на одном конце, называемом вершиной стека.
2. Реализация стека использует связный список, где каждый элемент списка содержит значение элемента и указатель на следующий элемент.
3. Добавление элемента в стек осуществляется путем добавления нового элемента в начало связного списка.
4. Удаление элемента из стека осуществляется путем удаления первого элемента связного списка.
5. Стек поддерживает операции, такие как проверка на пустоту, определение размера стека, доступ к вершине стека, поиск элемента в стеке и подсчет количества определенных элементов в стеке.
6. Реализация стека также предоставляет операторы присваивания и перемещения для удобства работы с объектами класса stack.
7. Для удобства тестирования реализации стека были созданы модульные тесты с использованием фреймворка Google Test.

Таким образом, реализация стека на основе связного списка предоставляет эффективную структуру данных для хранения и обработки элементов в порядке Last-In-First-Out (LIFO). Это полезная структура данных, применимая во многих задачах, где требуется временное хранение элементов с возможностью их быстрого добавления и удаления.

### СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.*

Алгоритмы: построение и анализ, 3-е издание = Introduction to Algorithms, Third Edition.

— М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=Вильямс_(издательство)&action=edit&redlink=1), 2013. — 1328 с. — ISBN 978-5-8459-1794-2

1. *Курносов М.Г., Берлизов Д.М.* Алгоритмы и структуры обработки информации. – Новосибирск: Параллель, 2019. – 211 с. — ISBN 978-5-98901-230-5
2. *Guttmann A.* R-trees: A dynamic index structure for spatial searching // ACM SIGMOD. - 1984. - №14(2). - С. 47-57.
3. *Samet H.* The design and analysis of spatial data structures. - 2 изд. - Addison-Wesley Publishing Co., 1990

### ПРИЛОЖЕНИЯ

**Приложение 1.** Исходный код проекта

1. Файл CMakeLists.txt

cmake\_minimum\_required(VERSION 3.22)

project(

"stack"

VERSION 1.0

LANGUAGES CXX

)

set(CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY ${CMAKE\_BINARY\_DIR}/bin)

list(APPEND CMAKE\_MODULE\_PATH ${CMAKE\_CURRENT\_LIST\_DIR}/cmake)

add\_subdirectory(src)

add\_subdirectory(test)

add\_subdirectory(external)

2. Файл list.hpp

#pragma once

#include <memory>

#include <new>

#include <utility>

#include <iostream>

template <typename T>

struct node {

node() : next(nullptr) {}

node(const T& val) : value(val), next(nullptr) {}

T value;

node<T>\* next;

};

namespace list {

template <typename T, typename allocator = std::allocator<node<T>>>

class list {

template <typename LT>

friend bool operator==(const list<LT>& lst1, const list<LT>& lst2);

public:

list();

list(list<T>&& l);

list<T>& operator=(list<T>& l);

list<T>& operator=(list<T>&& l);

virtual ~list();

list(const std::initializer\_list<T>& init);

list(const list<T>& l);

bool is\_empty() const;

size\_t size() const;

node<T>\* begin() const;

node<T>\* end() const;

const node<T>\* cbegin() const;

const node<T>\* cend() const;

void push\_back(const T& value);

void push\_front(const T& value);

void pop\_back();

void pop\_front();

list<T>& erase(const size\_t idx);

list<T>& erase\_first(const T& v);

list<T>& erase\_last(const T& v);

list<T>& unique();

bool contains(const T& v) const;

size\_t count(const T& v) const;

node<T>& operator[](const size\_t idx);

const node<T>& operator[](const size\_t idx) const;

void clear();

void copy(const list<T>& l);

void merge(const list<T>& l);

void merge(list<T>&& l);

private:

node<T>\* head = nullptr;

node<T>\* tail = nullptr;

size\_t size\_ = 0;

std::allocator<node<T>> allctr;

inline node<T>\* alloc\_node(const T& value) {

node<T>\* newnode = allctr.allocate(1);

new (newnode) node<T>(value);

return newnode;

}

};

/\* Standard constructor. \*/

template <typename T, typename allocator>

list<T, allocator>::list() : head(nullptr), tail(nullptr), size\_(0) {}

/\* std::initializer\_list constructor. \*/

template <typename T, typename allocator>

list<T, allocator>::list(const std::initializer\_list<T>& init)

: head(nullptr), tail(nullptr), size\_(init.size()) {

if (init.size() != 0) {

node<T>\* new\_node = new node<T>(\*init.begin());

head = tail = new\_node;

for (auto c = init.begin() + 1; c != init.end(); ++c) {

new\_node = new node<T>(\*c);

tail->next = new\_node;

tail = new\_node;

}

}

}

/\* Copy constructor. \*/

template <typename T, typename allocator>

list<T, allocator>::list(const list<T>& l) {

copy(l);

}

/\* Move constructor. \*/

template <typename T, typename allocator>

list<T, allocator>::list(list<T>&& l) {

std::swap(head, l.head);

std::swap(tail, l.tail);

std::swap(size\_, l.size\_);

l.tail = l.head = nullptr;

l.size\_ = 0;

}

/\* Copy assignment operator. \*/

template <typename T, typename allocator>

list<T>& list<T, allocator>::operator=(list<T>& l) {

clear();

copy(l);

return \*this;

}

/\* Move assingment operator. \*/

template <typename T, typename allocator>

list<T>& list<T, allocator>::operator=(list<T>&& l) {

std::swap(head, l.head);

std::swap(tail, l.tail);

std::swap(size\_, l.size\_);

return \*this;

}

/\* Destructor. \*/

template <typename T, typename allocator>

list<T, allocator>::~list() {

clear();

}

/\* Returns true if list is empty. \*/

template <typename T, typename allocator>

bool list<T, allocator>::is\_empty() const {

return (size\_ == 0);

}

/\* Returns number of elements in the list. \*/

template <typename T, typename allocator>

size\_t list<T, allocator>::size() const {

return size\_;

}

/\* Returns pointer to the first element in the list. \*/

template <typename T, typename allocator>

node<T>\* list<T, allocator>::begin() const {

return head;

}

/\* Returns pointer to the last element in the list. \*/

template <typename T, typename allocator>

node<T>\* list<T, allocator>::end() const {

return tail;

}

/\* Returns constant pointer to the first element in the list. \*/

template <typename T, typename allocator>

const node<T>\* list<T, allocator>::cbegin() const {

return const\_cast<const node<T>\* const>(head);

}

/\* Returns constant pointer to the last element in the list. \*/

template <typename T, typename allocator>

const node<T>\* list<T, allocator>::cend() const {

return const\_cast<const node<T>\* const>(tail);

}

/\* Add element to the end of the list. \*/

template <typename T, typename allocator>

void list<T, allocator>::push\_back(const T& value) {

node<T>\* new\_node = alloc\_node(value);

if (is\_empty()) [[unlikely]] {

tail = head = new\_node;

++size\_;

return;

}

tail->next = new\_node;

tail = new\_node;

++size\_;

}

/\* Removes element from the end of the list.

\* Returns void. \*/

template <typename T, typename allocator>

void list<T, allocator>::pop\_back() {

if (is\_empty()) {

return;

}

node<T>\* prev = nullptr;

node<T>\* deletable = head;

while (deletable != nullptr) {

prev = deletable;

deletable = deletable->next;

}

if (prev == nullptr) {

allctr.deallocate(head, sizeof(\*head));

head = nullptr;

return;

}

tail = prev;

tail->next = nullptr;

allctr.deallocate(deletable, sizeof(\*deletable));

--size\_;

}

/\* Adds element to the front of the list.

\* Returns void. \*/

template <typename T, typename allocator>

void list<T, allocator>::push\_front(const T& value) {

node<T>\* new\_front\_val = alloc\_node(value);

node<T>\* temp\_head = head;

if (is\_empty()) {

++size\_;

head = tail = new\_front\_val;

return;

}

head = new\_front\_val;

head->next = temp\_head;

++size\_;

}

/\* Removes element from the front of the list.

\* Returns void. \*/

template <typename T, typename allocator>

void list<T, allocator>::pop\_front() {

if (is\_empty()) {

return;

}

node<T>\* deletable = head;

head = head->next;

allctr.deallocate(deletable, sizeof(\*deletable));

--size\_;

}

/\* Removes the first encountered elements with value v.

\* Returns reference to the current list. \*/

template <typename T, typename allocator>

list<T>& list<T, allocator>::erase(const size\_t idx) {

if (is\_empty()) {

return \*this;

}

if (idx == 0) {

pop\_front();

return \*this;

}

node<T>\* temp\_head = &(\*this)[idx - 1];

if (temp\_head->next == nullptr) {

return \*this;

}

node<T>\* deletable = temp\_head->next;

temp\_head->next = temp\_head->next->next;

allctr.deallocate(deletable, sizeof(\*deletable));

--size\_;

return \*this;

}

/\* Removes the first encountered elements with value v.

\* Returns reference to the current list. \*/

template <typename T, typename allocator>

list<T>& list<T, allocator>::erase\_first(const T& v) {

if (is\_empty()) {

return \*this;

}

if (head->value == v) {

pop\_front();

return \*this;

}

node<T>\* temp\_head = head;

while (temp\_head->next->value != v) {

temp\_head = temp\_head->next;

}

if (temp\_head->next == nullptr) {

return \*this;

}

node<T>\* deletable = temp\_head->next;

temp\_head->next = temp\_head->next->next;

allctr.deallocate(deletable, sizeof(\*deletable));

--size\_;

return \*this;

}

/\* Removes the last encountered elements with value v.

\* Returns reference to the current list. \*/

template <typename T, typename allocator>

list<T>& list<T, allocator>::erase\_last(const T& v) {

if (is\_empty()) {

return \*this;

}

if ((head->value == v && count(v) == 1)) {

pop\_front();

return \*this;

}

node<T>\* temp\_head = head;

node<T>\* nbeforedeletable = nullptr;

while (temp\_head->next != nullptr) {

if (temp\_head->next->value == v) {

nbeforedeletable = temp\_head;

}

temp\_head = temp\_head->next;

}

if (nbeforedeletable == nullptr) {

return \*this;

}

node<T>\* deletable = nbeforedeletable->next;

nbeforedeletable->next = nbeforedeletable->next->next;

allctr.deallocate(deletable, sizeof(\*deletable));

--size\_;

return \*this;

}

/\* Removes all consecutive duplicate elements from the container.

\* Returns reference to the current list. \*/

template <typename T, typename allocator>

list<T>& list<T, allocator>::unique() {

node<T>\* temp\_head = head;

size\_t idx\_counter = 0;

while (temp\_head->next != nullptr) {

if (temp\_head->value == temp\_head->next->value) {

erase(idx\_counter + 1);

continue;

}

++idx\_counter;

temp\_head = temp\_head->next;

}

return \*this;

}

/\* Checks if an element is in the list.

\* Returns 1 if element's containing. \*/

template <typename T, typename allocator>

bool list<T, allocator>::contains(const T& v) const {

if (is\_empty()) {

return false;

}

node<T>\* find = head;

while (find != nullptr) {

if (find->value == v) {

return true;

}

find = find->next;

}

return false;

}

/\* Return number of elements with value v in list. \*/

template <typename T, typename allocator>

size\_t list<T, allocator>::count(const T& v) const {

size\_t count = 0;

node<T>\* temp\_head = head;

while (temp\_head != nullptr) {

if (temp\_head->value == v) {

++count;

}

temp\_head = temp\_head->next;

}

return count;

}

/\* Return reference to element in list[idx]. \*/

template <typename T, typename allocator>

node<T>& list<T, allocator>::operator[](const size\_t idx) {

node<T>\* temp\_head = head;

size\_t beg = 0;

while (beg != idx && temp\_head != nullptr) {

temp\_head = temp\_head->next;

++beg;

}

return \*temp\_head;

}

/\* Return const reference to element in list[idx]. \*/

template <typename T, typename allocator>

const node<T>& list<T, allocator>::operator[](const size\_t idx) const {

node<T>\* temp\_head = head;

size\_t beg = 0;

while (beg != idx && temp\_head != nullptr) {

temp\_head = temp\_head->next;

++beg;

}

return \*temp\_head;

}

/\* Similar to ~list(). \*/

template <typename T, typename allocator>

void list<T, allocator>::clear() {

while (head != nullptr) {

node<T>\* deletable = std::exchange(head, head->next);

allctr.deallocate(deletable, sizeof(\*deletable));

}

}

/\* Copies l to the current list. \*/

template <typename T, typename allocator>

void list<T, allocator>::copy(const list<T>& l) {

if (l.head != nullptr) {

node<T>\* iter = l.head;

while (iter != nullptr) {

push\_back(iter->value);

iter = iter->next;

}

size\_ = l.size\_;

}

}

/\* Add l to the end of the current list. \*/

template <typename T, typename allocator>

void list<T, allocator>::merge(const list<T>& l) {

node<T>\* iter = l.head;

while (iter) {

push\_back(iter->value);

iter = iter->next;

}

}

/\* Add l to the end of the current list. \*/

template <typename T, typename allocator>

void list<T, allocator>::merge(list<T>&& l) {

std::swap(tail->next, l.head);

std::swap(tail, l.tail);

size\_ += l.size\_;

l.head = l.tail = nullptr;

l.size\_ = 0;

}

// Just for testing.

template <typename T>

bool operator==(const list<T>& lst1, const list<T>& lst2) {

if (lst1.size() != lst2.size()) {

return false;

}

for (auto cntr1 = lst1.head, cntr2 = lst2.head;

cntr1 != nullptr && cntr2 != nullptr;

cntr1 = cntr1->next, cntr2 = cntr2->next) {

if (cntr1->value != cntr2->value) {

return false;

}

}

return true;

}

}; // namespace list

3. Файл stack.hpp

#pragma once

#include "../liblist/list.hpp"

namespace stack {

template <typename T>

class stack : private list::list<T> {

template <typename U>

friend bool operator==(const stack<U>& lst1, const stack<U>& lst2);

public:

stack() = default;

stack(const stack<T>& s) = default;

stack(stack<T>&& s) = default;

stack& operator=(const stack<T>& s);

stack& operator=(stack<T>&& s);

~stack() = default;

stack(const std::initializer\_list<T>& init\_list);

size\_t size() const;

bool is\_empty() const;

node<T>\* begin() const;

node<T>\* end() const;

const node<T>\* cbegin() const;

const node<T>\* cend() const;

void push(const T& x);

void pop();

bool contains(const T& value) const;

size\_t count(const T& v) const;

void print\_stack(std::ostream& os) const;

};

/\* std::initializer\_list constructor. \*/

template <typename T>

stack<T>::stack(const std::initializer\_list<T>& init\_list)

{

for (const auto& c : init\_list) {

list::list<T>::push\_front(c);

}

}

/\* Copy assignment operator. \*/

template <typename T>

stack<T>& stack<T>::operator=(const stack<T>& s)

{

this->clear();

this->copy(s);

return \*this;

}

/\* Move assingment operator. \*/

template <typename T>

stack<T>& stack<T>::operator=(stack<T>&& s)

{

return this->operator=(s);

}

/\* Returns pointer to the first element in the stack. \*/

template <typename T>

node<T>\* stack<T>::begin() const

{

return list::list<T>::begin();

}

/\* Returns pointer to the last element in the stack. \*/

template <typename T>

node<T>\* stack<T>::end() const

{

return list::list<T>::end();

}

/\* Returns constant pointer to the first element in the stack. \*/

template <typename T>

const node<T>\* stack<T>::cbegin() const

{

return list::list<T>::cbegin();

}

/\* Returns constant pointer to the last element in the stack. \*/

template <typename T>

const node<T>\* stack<T>::cend() const

{

return list::list<T>::cend();

}

/\* Add element to the stack. \*/

template <typename T>

void stack<T>::push(const T& value)

{

list::list<T>::push\_front(value);

}

/\* Removes element from stack. \*/

template <typename T>

void stack<T>::pop()

{

list::list<T>::pop\_front();

}

/\* Checks if an element is in the stack.

\* Returns 1 if element's containing. \*/

template <typename T>

bool stack<T>::contains(const T& value) const

{

return list::list<T>::contains(value);

}

/\* Return number of elements with value v in stack. \*/

template <typename T>

size\_t stack<T>::count(const T& v) const

{

return list::list<T>::count(v);

}

/\* Returns number of elements in the stack. \*/

template <typename T>

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

{

return this->list::list<T>::size();

}

/\* Returns true if stack is empty. \*/

template <typename T>

bool stack<T>::is\_empty() const

{

return this->list::list<T>::is\_empty();

}

// Just for testing.

template <typename T>

bool operator==(const stack<T>& lst1, const stack<T>& lst2)

{

return list::operator==(lst1, lst2);

}

} // namespace stack

**Отзыв о работе студента** Киреев Илья Анатольевич (ФИО студента)

# Уровень освоения компетенций

Киреев Илья Анатольевич

(ФИО студента)

|  |  |
| --- | --- |
| Компетенции | Уровень сформированности компетенций |
| *ОПК-1 - Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной деятельности* |  |

## отметка о зачете

Руководитель практики от СибГУТИ:

Должность руководителя подпись ФИО руководителя

## " " 20 г.