МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации



**ЛАБОРАТОРНАЯ РАБОТА №1**

**по дисциплине: «Программирование»**

**на тему: «Абстрактные структуры данных»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АИ-42», «АВТФ»,  *Зиновьев Александр Евгеньевич*  «\_\_\_» \_\_\_\_\_\_ 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Ассистент кафедры ЗИ*  *Исаев Глеб Андреевич*  «\_\_\_» \_\_\_\_\_\_ 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc196301942)

[Ход выполнения 4](#_Toc196301943)

[Вывод](#_Toc196301944) 9

# ВВЕДЕНИЕ

1. Реализовать структуры данных с базовым набором операций:

a. Массив. Операции: создание, добавление элемента (по индексу и в конец массива), получение элемента по индексу, удаление элемента по индексу, замена элемента по индексу, длина массива, чтение.

b. Односвязный список. Операции: добавление и удаление элемента (4 способа (до после голова хвост)), чтение (разное, несколько способов), удаление элемента по значению, поиск элемента по значению.

c. Двусвязный список. Операции: добавление и удаление элемента (4 способа (до после голова хвост)), чтение (разное, несколько способов), удаление элемента по значению, поиск элемента по значению.

d. Стек. Операции: добавление и удаление элемента (push и pop), чтение.

e. Очередь. Операции: добавление и удаление элемента (push и pop), чтение.

f. Деревья (вариант 11 Красно-черное дерево, вариант 2 - Сбалансированное двоичное дерево (АВЛ-дерево), вариант 3 Complete Binary Tree, вариант 4 Full Binary Tree, вариант 5 – бинарное дерево поиска. Операции: добавление элемента, поиск элемента, удаление элемента с сохранением структуры дерева, чтение. Если Full Binary Tree или Complete Binary Tree, то вместо удаления элемента добавить операцию проверки на Full или Complete соответственно.

2. Реализовать интерфейс работы со структурами данных. Предусмотреть считывание из файла и запись в файл при внесении изменений в состав элементов. Часть операций для реализации представлено в таблице. Дополнить список операций согласно п.1 и сохраняя стиль написания команды (первая буква – принадлежность структуре данных). Исключение: Операция PRINT выводит любую структуру данных на экран.

3. Обосновать сложность выполнения каждой операции с позиции BigO нотации. Составить таблицу BigO по операциям и структурам в отчете.

4. Представить достоинства и недостатки каждой структуры данных, описать

сферу применения (примеры).

5. В отчете в контрольном примере представить визуальные формы, подтверждающие соответствие результатам работы программы.

**Весь код находится в репозитории:**

ХОД ВЫПОЛНЕНИЯ

Задание 1.

Реализацию абстрактных структур данных я начал с структуры односвязного списка (Forward List), у которого каждый узел «знает» хранит в себе своё значение и указатель на следующей узел. Поэтому я соответственно в заголовочном файле создал структуру Node содержащую value и next(рисунок 1).

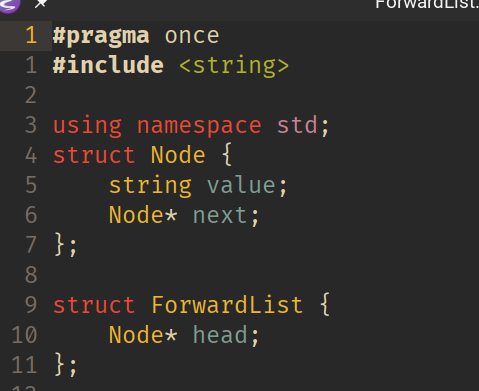
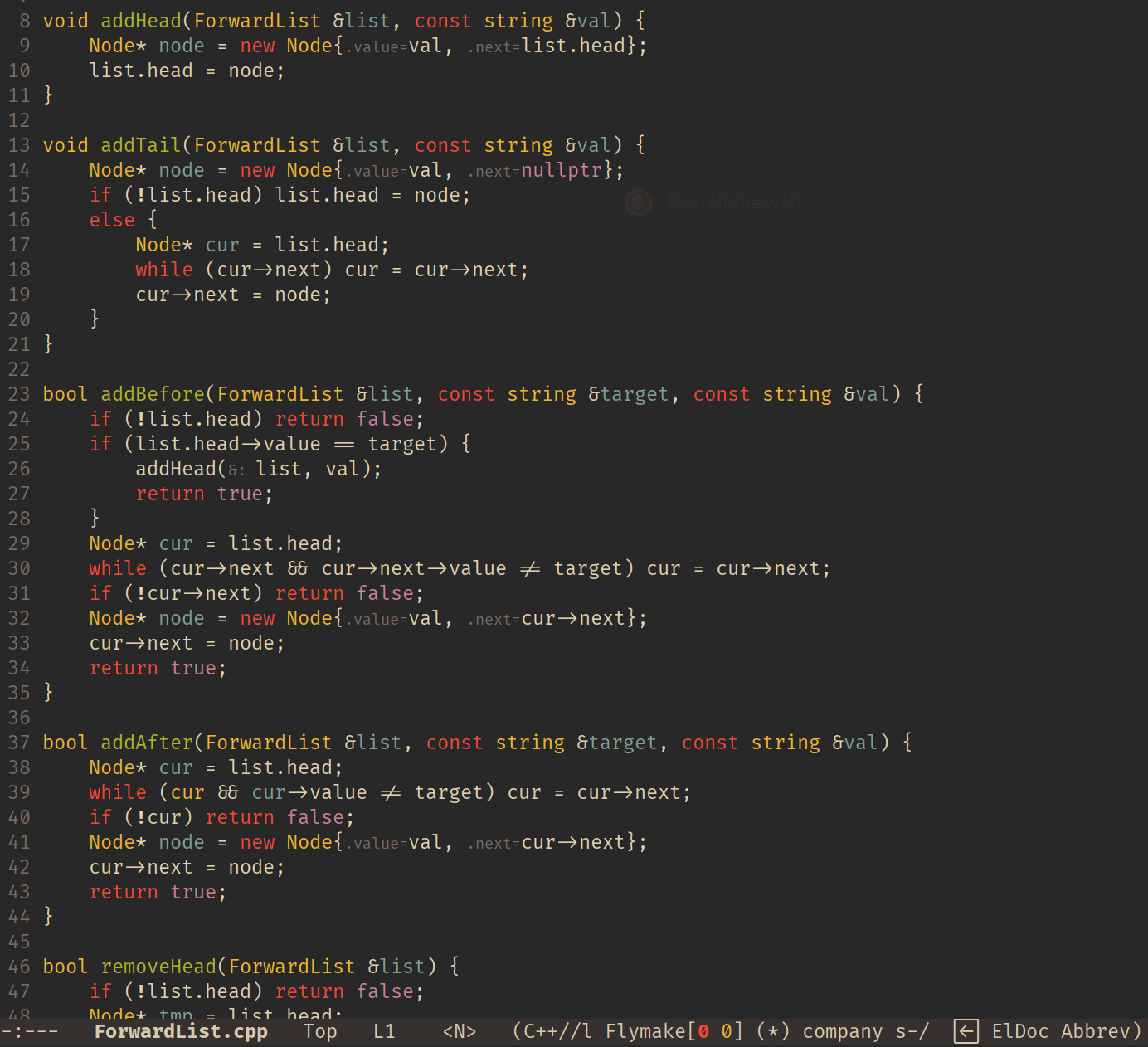
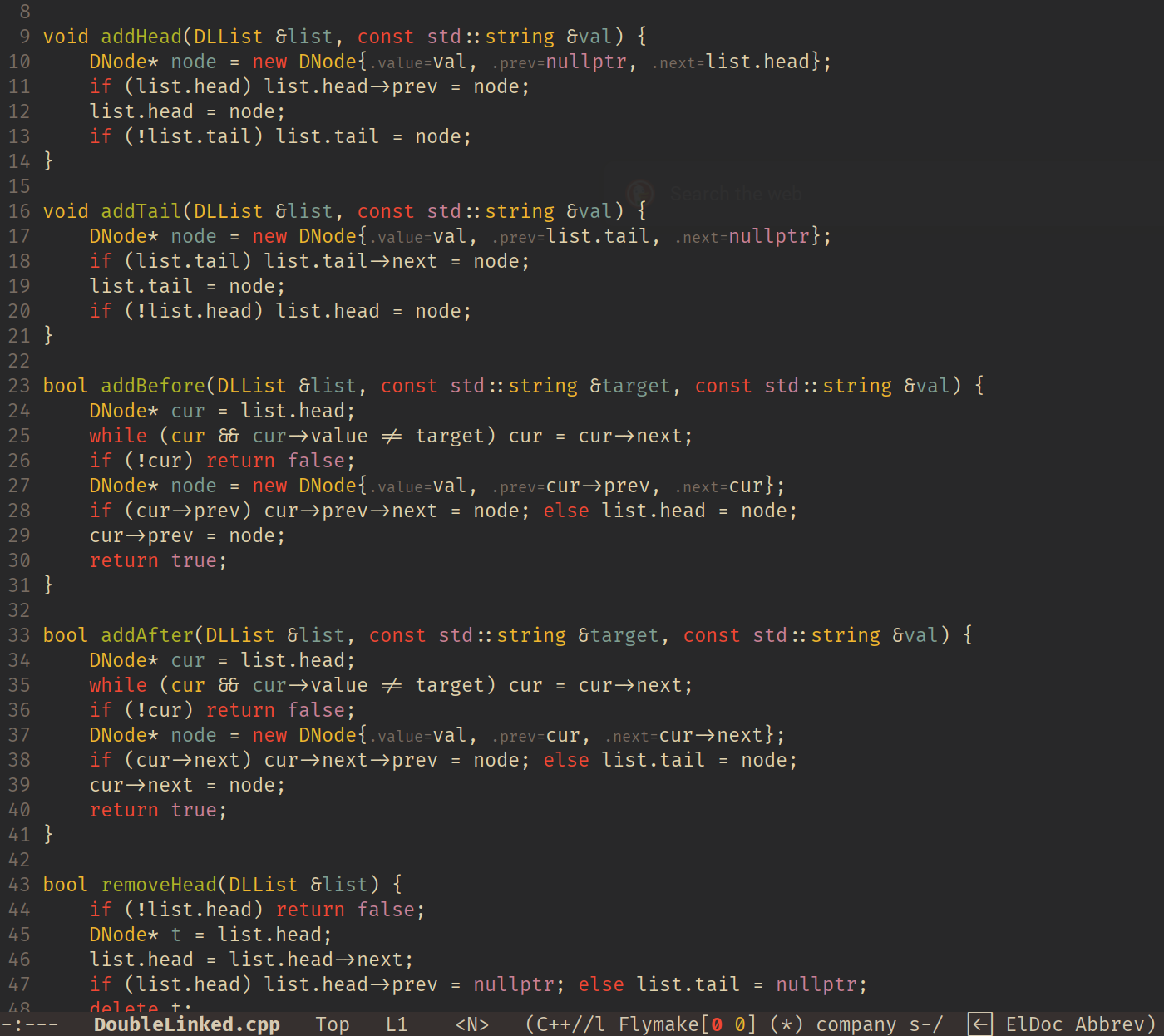


Рисунок 1 — Объявление структуры в ForwardList.h

После чего добавил основные операции, например, добавления до/после определённого узла, в начало конец, аналогично для удаления(рисунок 2).

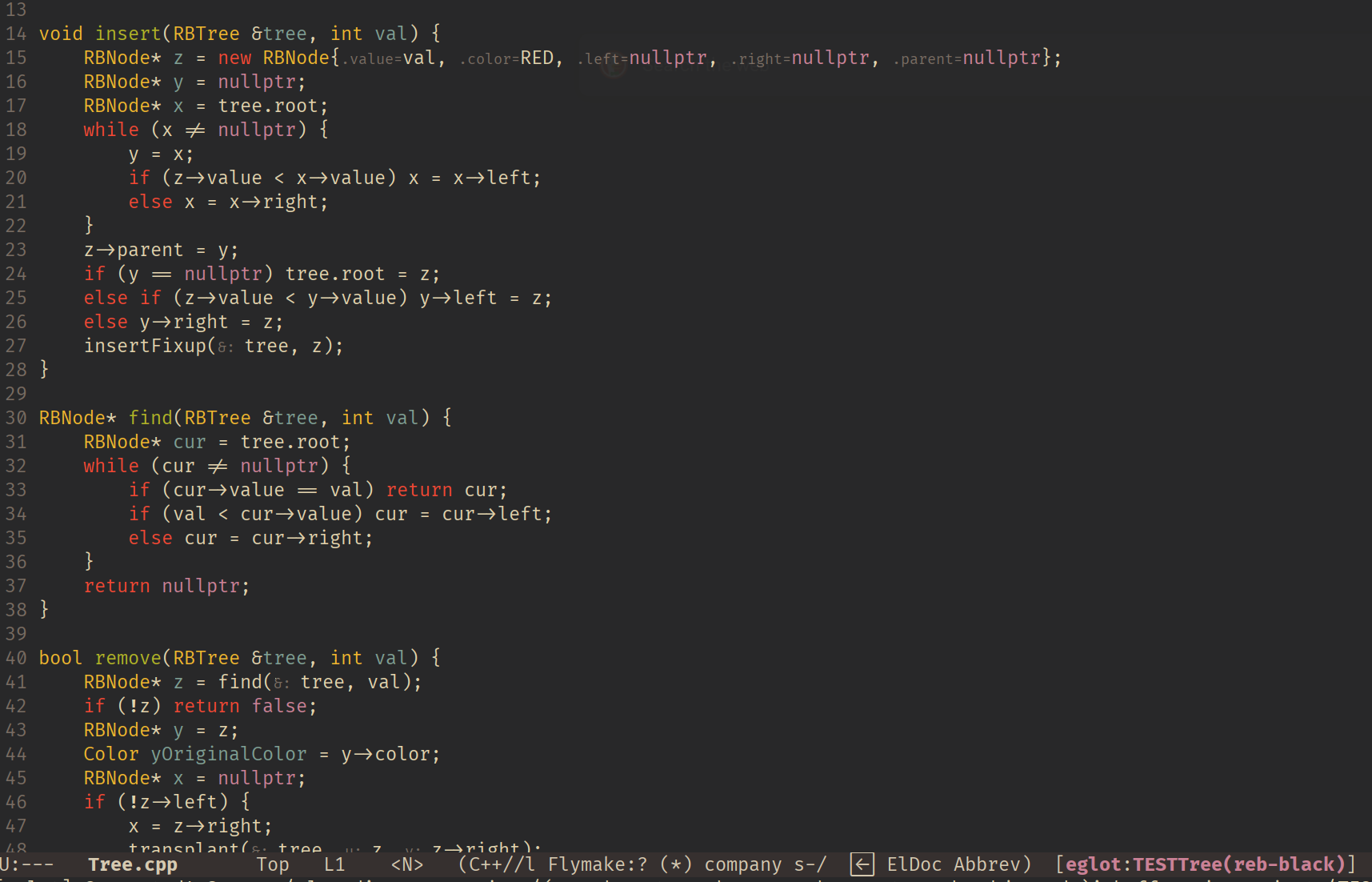
Рисунок 2 — Часть реализаций функций Forward List

После полной реализации односвязного списка, я приступил к двусвязному. Так как реализация двусвязного опирается на односвязный, то весь код является дополненной версией односвязного списка (рисунок 3).

Рисунок 3 — Часть реализаций функций для Double linked list

Потом я приступил к массиву, стеку, по аналогии со стэком — очереди, и, наконец, чёрно-красному белому (рисунок 4-5)

Рисунок 4 — Часть реализаций функций в Array

Рисунок 5 — Часть реализаций функций в Tree(red-black)

Полная реализация всех АСД находится в репозитории github.

Задание 2.

В процессе выполнения задания я к каждой структуре АСД добавил файл — Interface.cpp, в котором находится main-файл, реализаующий интерактивный ввод операций для каждой АСД, пример для односвязного списка на рисунке 6.

Рисунок 6 — Интерактивный ввод операций для Forward List

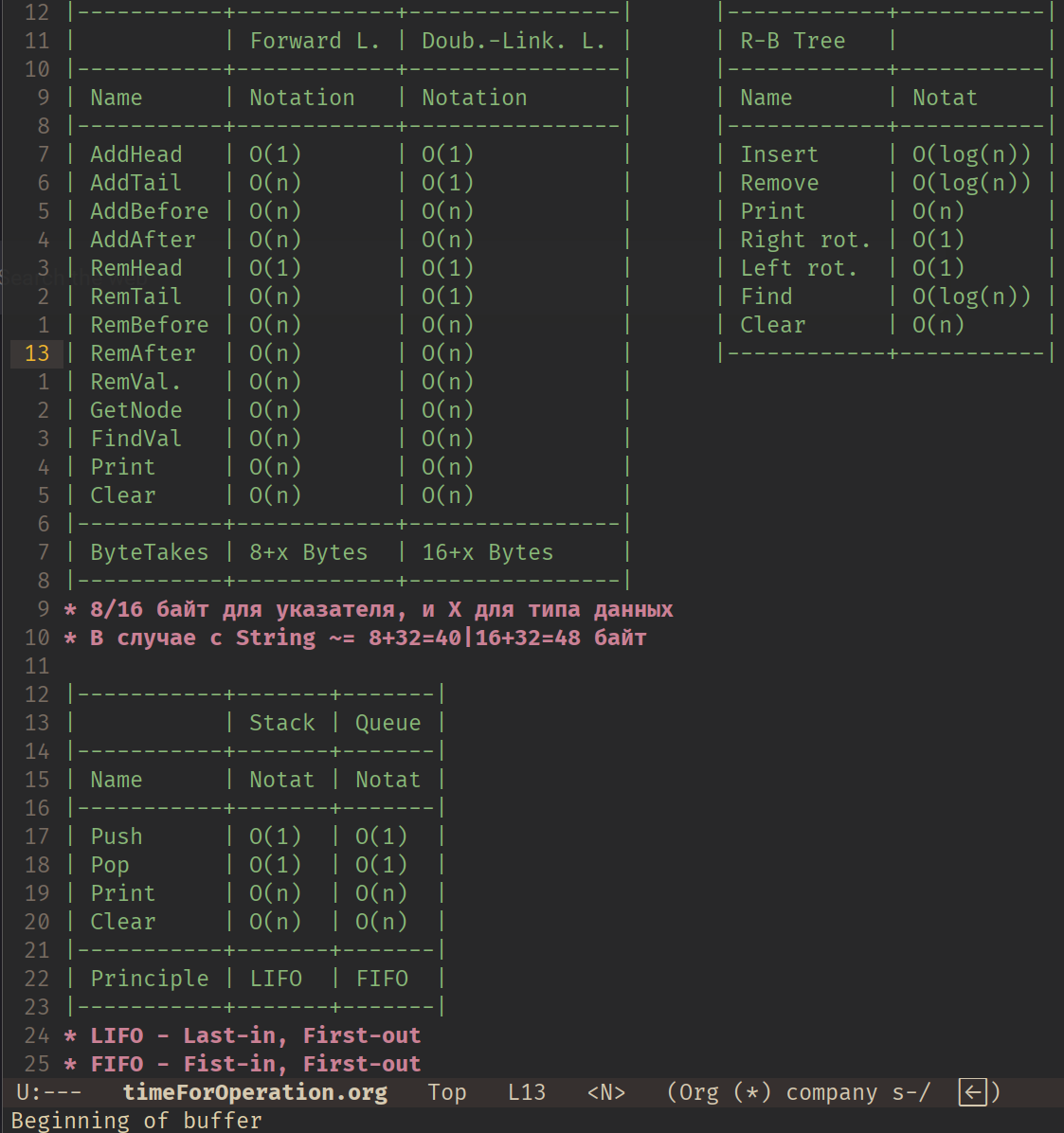
Задание 3

В большинстве операций односвязного и двусвязного списка сложность операций выполнения операций занимает O(n) так как они требуют прохождения с начала (голова, head) списка до конца (хвоста, tail). Именно поэтому операции такие как: Чтение, добавление до/после, добавление в конец(для односвязного), удаление до/после, удаление в конце(для односвязного) занимают O(n), в то время как добавление(удаления) в начало, конец(для двусвязного) занимают O(1) потому, что в массиве хранится адрес начала односвязного списка (и конца для двусвязного списка).

В стеке и очереди основные операции (push/pop) занимают O(1), так как стек имеет адрес вершины в которую мы можем поместить значение (push) или достать (pop), в соответствии с LIFO (Last-In, Fisrt-Out), аналогично для очереди, с тем отличием, что очередь имеет адрес самого первого элемента, который достаётся (pop) во время работы, в соотвествии с FIFO(First-In, Fisrt-Out).

В красно-черном дереве (Red-Black Tree) сложность основных операций (вставка, удаление, поиск) составляет O(log n), где n - количество элементов в дереве. Это обеспечивается за счет строгих правил балансировки: каждый узел имеет цвет (красный или черный), корень всегда черный, красные узлы не могут иметь красных потомков, и все пути от любого узла до листьев содержат одинаковое количество черных узлов. Эти свойства гарантируют, что высота дерева не превышает 2log₂(n+1), что обеспечивает логарифмическое время выполнения операций даже в худшем случае. При вставке или удалении выполняются перекрашивания и вращения (O(1)), но в худшем случае может потребоваться пройти от листа до корня (O(log n)) для восстановления свойств дерева.

Таблица со сложностью выполнения находится в репозитории, где находится и весь код, а так же на рисунке 7. Таблица timeForOperation.org совместима с org-mode текствого редактора Emacs.

Рисунок 7 — Таблица со сложностью операций для всех АСД

Задание 4.

Односвязный список обладает достоинством эффективного использования памяти, поскольку каждый узел требует минимального количества дополнительной памяти - только один указатель на следующий элемент. Его основные операции вставки и удаления в начале списка выполняются за постоянное время O(1), что делает его идеальным для реализации стеков и очередей. Однако он имеет серьезный недостаток - необходимость последовательного доступа к элементам, поскольку произвольный доступ требует O(n) времени в худшем случае. Односвязные списки находят применение в системах, где важна экономия памяти и часто выполняются операции в начале списка, таких как кэширование недавно использованных элементов или управление историей операций.

Двусвязный список решает основную проблему односвязного списка, обеспечивая двунаправленный обход за счет хранения двух указателей в каждом узле. Это позволяет эффективно выполнять операции как в начале, так и в конце списка, а также удалять элементы без необходимости знать предыдущий узел. К недостаткам можно отнести повышенное потребление памяти и более сложную реализацию операций модификации. Двусвязные списки широко используются в системах навигации, текстовых редакторах, кэшах типа LRU(2 лаба) и любых приложениях, требующих двунаправленного обхода данных.

Стек демонстрирует превосходную производительность для операций LIFO, обеспечивая постоянное время выполнения для push и pop операций. Его простота реализации и минимальные накладные расходы делают его исключительно эффективным для управления вызовами функций и контроля памяти. Основной недостаток стека - ограниченный доступ к данным, поскольку можно работать только с верхним элементом. Стеки незаменимы в компиляторах для разбора выражений, системах отмены действий в приложениях, управлении вызовами функций в языках программирования и алгоритмах обхода графов.

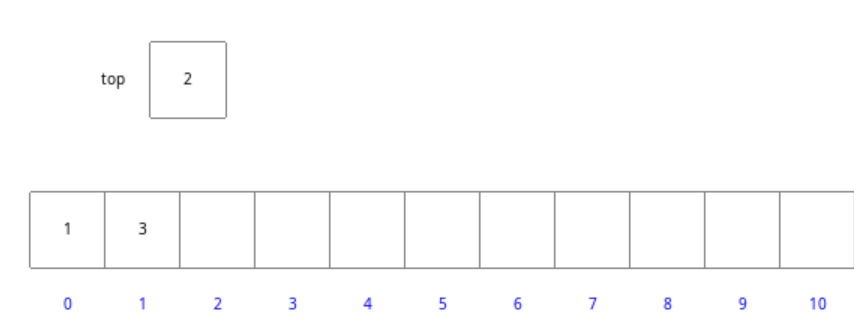
Очередь обеспечивает эффективную организацию данных по принципу FIFO с постоянным временем выполнения основных операций. Ее структура идеально подходит для обработки задач в порядке их поступления и буферизации данных. Как и стек, очередь ограничивает доступ к элементам, позволяя работать только с началом и концом последовательности. Очереди находят широкое применение в системах обработки сообщений, планировщиках задач операционных систем, сетевых буферах и моделировании реальных процессов, таких как обслуживание клиентов.

Красно-чёрное дерево сочетает преимущества бинарных деревьев поиска с гарантированной сбалансированностью, обеспечивая логарифмическое время выполнения операций даже в худшем случае. Его самобалансирующаяся природа исключает вырождение в связанный список, что является распространенной проблемой обычных BST. К недостаткам можно отнести сложность реализации и повышенные накладные расходы на хранение дополнительной информации о цвете узлов. Красно-чёрные деревья широко используются в базах данных, файловых системах, реализациях ассоциативных массивов в стандартных библиотеках (таких как std::map в C++ и TreeMap в Java), и любых приложениях, требующих эффективного динамического поддержания упорядоченных данных.

Задание 5

STACK

push — добавляем значения (рисунок 8,9)

Рисунок 8 — Визуализация добавления в стэк

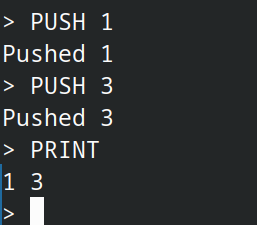
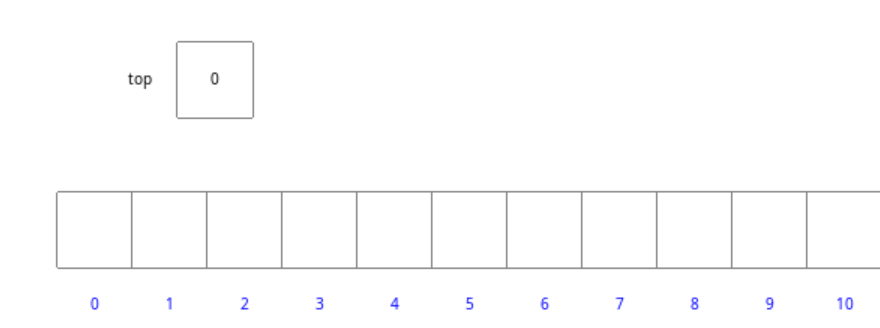


Рисунок 9 — Добавление в стэк, вывод стэка

pop — удаление последнего значения (рисунок 10, 11)

Рисунок 10 — Визуализация удаления из стэка

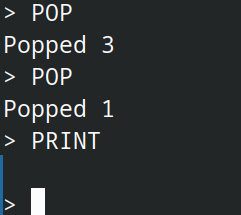
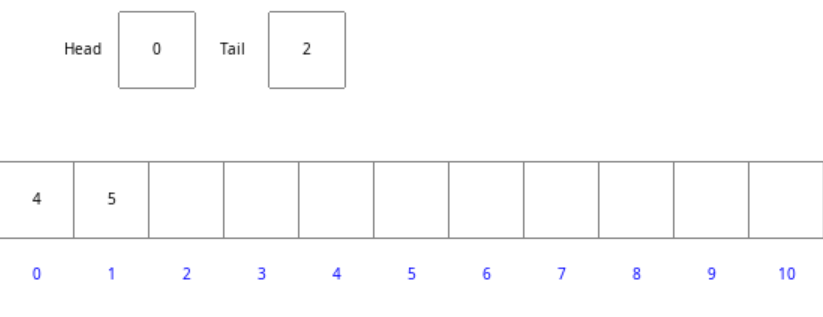


Рисунок 11 — Удаление из стэка, вывод стэка

Print — вывод стэка (рисунок 9, 11).

QUEUE

push — добавление в очередь (рисунок 12,13).

Рисунок 12 — Визуализация добавления в очередь

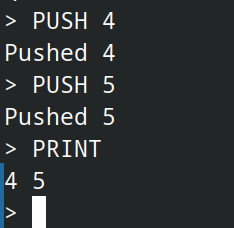
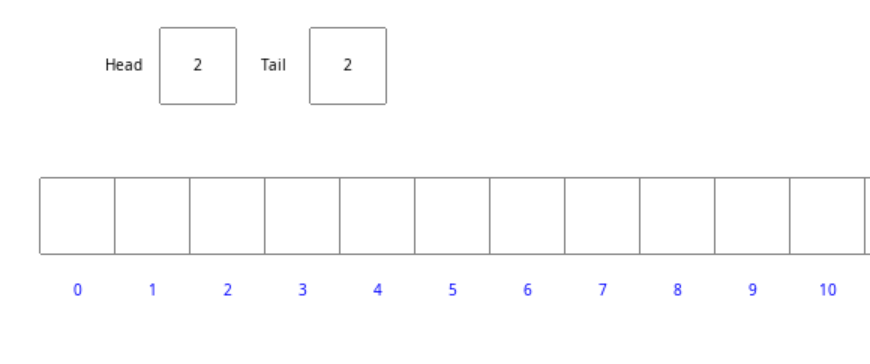


Рисунок 13 — Добавление в очередь

pop — удаление первого из очереди (рисунок 14,15).

Рисунок 14 — Визуализация удаления из очереди

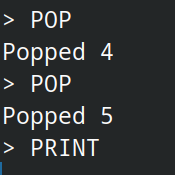


Рисунок 15 — удаление из очереди

print — вывод очереди (рисунок 13, 15).

Forward List

LADDH — Добавление элемента в head (рисунок 16, 17)

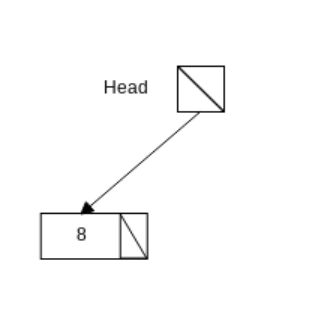


Рисунок 16 — Визуализация добавления в head

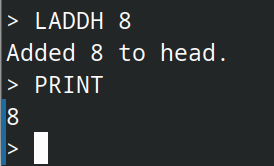


Рисунок 17 — Добавление в head

LADDT — Добавление в конец (рисунок 18, 19)

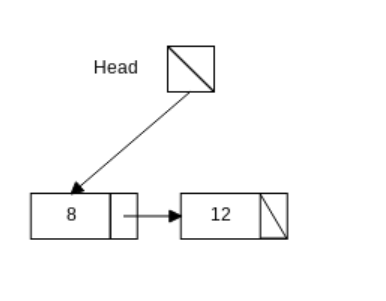


Рисунок 18 — Визулизация Добавления в конец

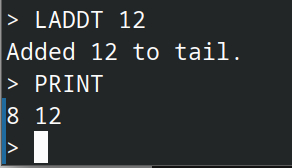


Рисунок 19 — Добавление в конец

LREMH — Удаление с начала (рисунок 20, 21)

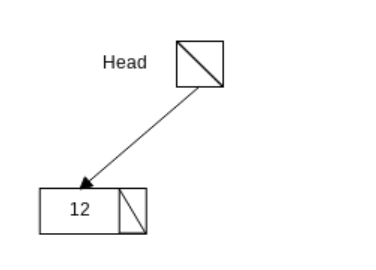


Рисунок 20 — Визуализация удаления начала

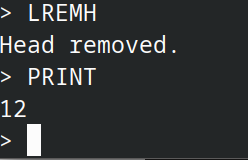


Рисунок 21 — Удаления с начала

LREMT — Удаление с конца (рисунок 22, 23)

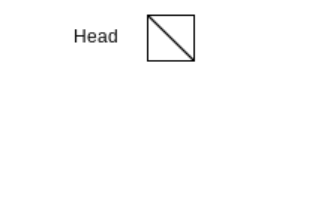


Рисунок 22 — Визуализация удаления с конца

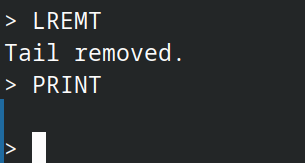


Рисунок 23 — Удаление с конца

Print — Вывод односвязного списка (рисунки 17, 19, 21, 23)

Double Linked List

DADDH — Добавление элемента в head (рисунок 24, 25)

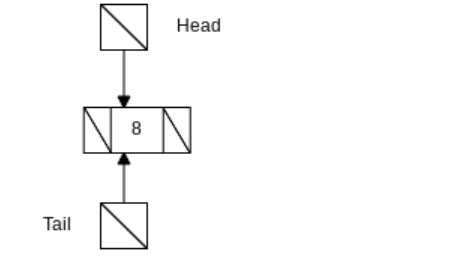


Рисунок 24 — Визуализация добавления в head

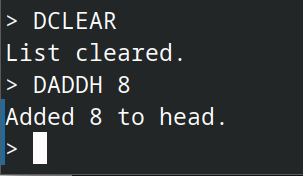


Рисунок 25 — Добавление в head

DADDT — Добавление в конец (рисунок 26, 27)

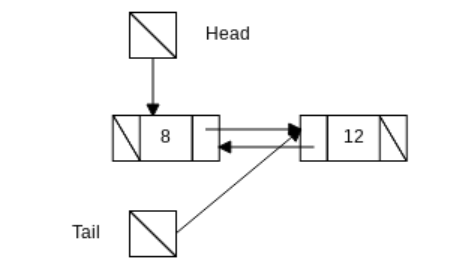


Рисунок 26 — Визулизация Добавления в конец

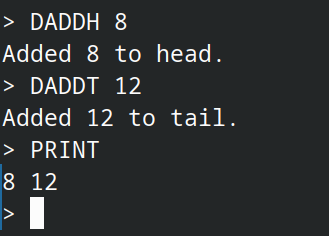


Рисунок 27 — Добавление в конец

DREMH — Удаление с начала (рисунок 28, 29)

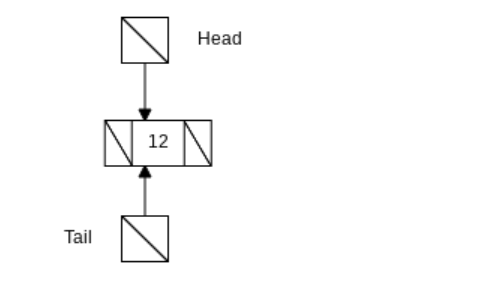


Рисунок 28 — Визуализация удаления начала

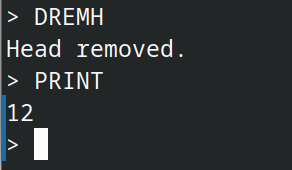


Рисунок 29 — Удаления с начала

DREMT — Удаление с конца (рисунок 30, 31)

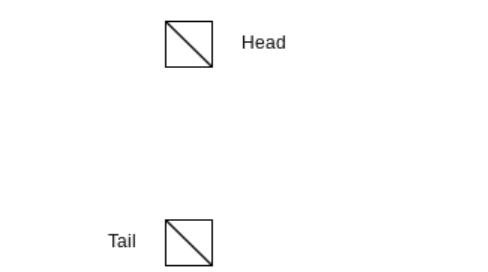


Рисунок 30 — Визуализация удаления с конца

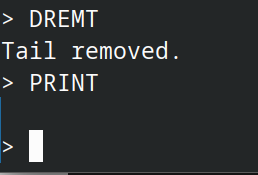
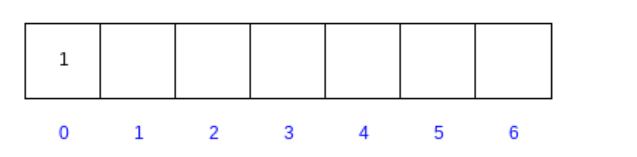


Рисунок 31 — Удаление с конца

Print — Вывод односвязного списка (рисунки 25, 27, 29, 31)

Array

AADD — добавить по индексу значение (рисунок 32, 33)

Рисунок 32 — Визуализация добавления в начало массива

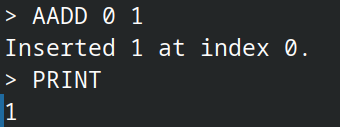


Рисунок 33 — Добавление в начало массива

ASET — изменить значение по индексу (рисунок 34, 35)

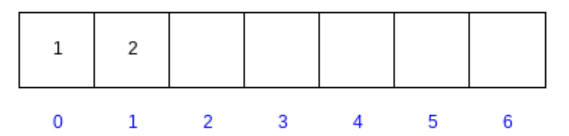


Рисунок 34 — Визуализация изменения значения по индексу

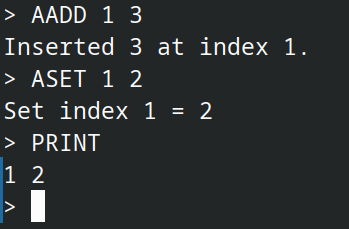


Рисунок 35 — Изменение значения по индексу

AREM — удаление по индексу(рисунок 36, 37)

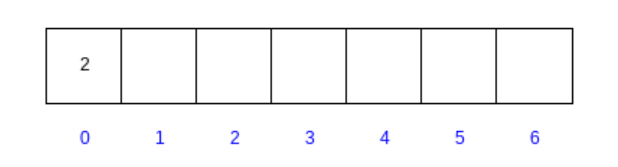


Рисунок 36 — Визуализация удаления по индексу

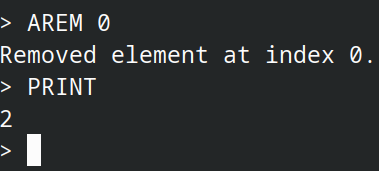


Рисунок 37 — Удаление по индексу

Print — выведение массива (рисунки 33, 35, 37)

Red-black tree

Insert — добавить элемент (рисунок 38, 39)

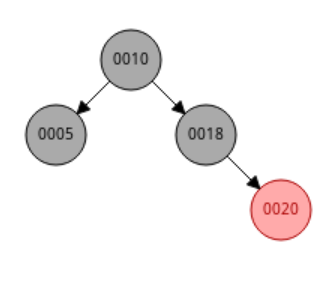


Рисунок 38 — Визуализация добавления элементов

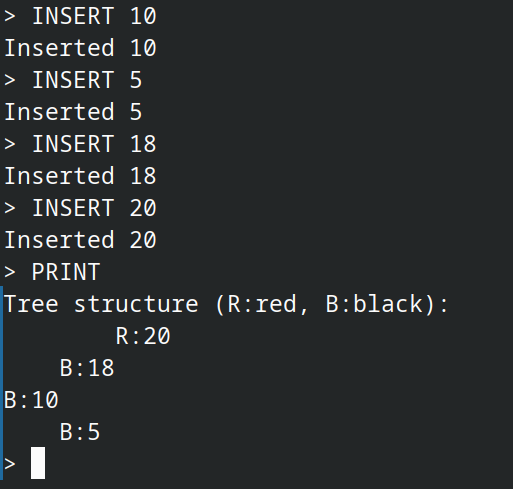


Рисунок 39 — Добавление элементов

REMOVE — удаление элемента (рисунок 40, 41)

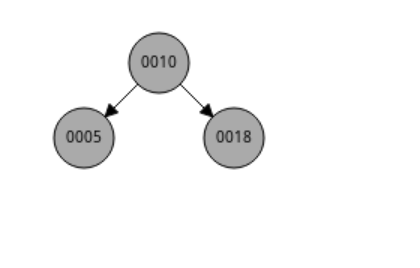


Рисунок 40 — Визуализация удаления элемента

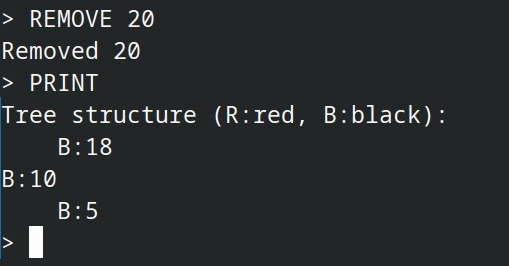


Рисунок 41 — Удаление элемента

print — выведение дерева (рисунок 39, 41)

ВЫВОД

В ходе выполнения работы была проведена комплексная реализация и анализ основных абстрактных структур данных. Реализованы массивы, односвязные и двусвязные списки, стеки, очереди и полное бинарное дерево с полным набором базовых операций. Каждая структура данных была протестирована на различных наборах исходных данных, что подтвердило корректность реализации алгоритмов.

Проведенный анализ временной сложности операций в нотации Big O показал существенные различия в эффективности различных структур данных при выполнении типовых операций. Массивы демонстрируют оптимальную производительность при операциях произвольного доступа, но уступают связным спискам при частых вставках и удалениях элементов. Связные списки, в свою очередь, обеспечивают эффективное выполнение операций модификации, но требуют линейного времени для операций доступа к произвольным элементам.

Разработанный интерфейс взаимодействия с пользователем обеспечивает единообразный доступ ко всем реализованным структурам данных через систему меню. Реализованные механизмы сохранения и загрузки данных из файлов позволяют обеспечить сохранность состояния структур данных между сеансами работы программы.

Сравнительный анализ структур данных выявил, что выбор оптимальной структуры должен определяться спецификой решаемой задачи.