МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Южно-Уральский государственный университет

(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук

Кафедра «Электронные вычислительные машины»

|  |  |
| --- | --- |
|  |  |

ОТчет по учебной практике

Алгоритмы и анализ сложности. Структуры данных

|  |  |
| --- | --- |
|  | Руководители учебной практики:  к.п.н., доцент каф. ЭВМ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ю.Г. Плаксина  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_2020 г.  старший преподаватель каф. ЭВМ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.Е. Беляков  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_2020 г.  Автор работы,  студент группы КЭ-105  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_В.Е.Урекина  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_2020 г. |

Челябинск-2020

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное   
образовательное учреждение высшего образования   
«Южно-Уральский государственный университет  
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук

Кафедра «Электронные вычислительные машины»

|  |  |
| --- | --- |
|  |  |

**Задание на учебную практику**

студенту группы КЭ-105

Урекиной Вере Евгеньевне

обучающемуся по направлению

09.03.01 «Информатика и вычислительная техника»

***Задание A8***

(Простые алгоритмы сортировки + анализ сложности по времени)

1) Написать три программы, реализующие простейшие алгоритмы сортировки: алгоритм сортировки пузырьком, алгоритм сортировки вставками и алгоритм сортировки выбором.

2) Экспериментально сравнить реализации этих алгоритмов по времени исполнения на различных наборах данных, различных по объему и различных по упорядоченности (обратить внимание на крайние случаи – уже отсортированный массив и массив, отсортированный в обратном порядке).

3) Сопоставить результаты с теоретическими оценками сложности алгоритмов (по времени исполнения). Сделать выводы о факторах, которых могли повлиять на точность измерений.

4) Сделать выводы, в каких случаях какой из алгоритмов является наиболее эффективным.

***Задание В8***

(Ассоциативный массив, хэш-таблица)

Ассоциативный массив (словарь) – структура данных, позволяющая хранить пары вида «(ключ, значение)», при этом предполагается уникальность ключа в массиве.

1) Разработать модуль (или библиотеку), который бы позволял хранить данные целого типа в массиве, индексами (ключами) которого являются строки.

2) Спроектировать ассоциативный массив таким образом, чтобы доступ до элемента выполнялся за время O(1), т.е. не зависел от количества элементов в массиве. Использовать в основе массива реализацию в виде хэш-таблицы. Предусмотреть возникновение коллизий.

3) Разработать отдельный модуль, который будет управлять массивом с помощью программы, считанной из файла.

В программе могут содержаться следующие команды:

* INSERT (“ключ”, значение) // добавить элемент в массив или заменить существующий
* FIND(“ключ”) // вывести значение или сообщить об его отсутствии
* REMOVE(“ключ”) // удалить элемент или сообщить об его отсутствии

4) При выполнении недопустимой команды выводить соответствующее сообщение об ошибке и переходить к следующей команде.

5) Входной текстовый файл содержит программу для управления массивом. 6) Результаты выполнения программы вывести на консоль.

**Срок сдачи студентом законченной работы (защита практики):** **27.07.2020.**

Руководители учебной практики \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/А.Е. Беляков

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Ю.Г. Плаксина

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/*В.Е.Урекина* /

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 5](#_Toc45722660)

[КРАТКОЕ ОПИСАНИЕ АЛГОРИТМА ЗАДАНИЕ А 11](#_Toc45722661)

[СХЕМА АЛГОРИТМА ЗАДАНИЯ А 12](#_Toc45722662)

[ЭКСПЕРИМЕНТАЛЬНЫЙ ГРАФИК ЗАВИСИМОСТИ ВРЕМЕНИ ВЫПОЛНЕНИЯ КОДА 15](#_Toc45722663)

[СОПОСТАВЛЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ С ТЕОРЕТИЧЕСКИМИ СЛОЖНОСТЯМИ АЛГОРИТМОВ (ПО ВРЕМЕНИ ИСПОЛНЕНИЯ) 18](#_Toc45722664)

[КРАТКОЕ ОПИСАНИЕ АЛГОРИТМА ЗАДАНИЯ В 21](#_Toc45722665)

[СХЕМА АЛГОРИТМА ЗАДАНИЯ В 22](#_Toc45722666)

[ОПИСАНИЕ РЕАЛИЗАЦИИ ЗАДАНИЯ В 25](#_Toc45722667)

[БИБЛИОГРАФИЧЕСКИЙ СПИСОК 27](#_Toc45722668)

[ПРИЛОЖЕНИЕ А 28](#_Toc45722669)

[ПРИЛОЖЕНИЕ В 33](#_Toc45722670)

# ВВЕДЕНИЕ

***Сложность алгоритмов***

Исследование сложности алгоритма помогает понять степень его практической приемлемости. Сравнительный анализ сложности нескольких алгоритмов решения одной и той же задачи позволяет сделать обоснованный выбор лучшего из них.

Слово «сложность» в этом контексте является математическим термином, а не общим обозначением препятствия к выполнению замысла. С понятие сложности связывают затраты времени и/или памяти, соответствующие худшему случаю , либо затраты в среднем[[1](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

Функцией сложности алгоритма считается О(f(N)), где f(N) − некоторая функция от исходных данных, если для любого N существует такая константа С, что время выполнения (или число операций) не превышает Cf(N). Функции перечислены в порядке возрастания сложности:

1. C–константа
2. log(log(N))
3. log(N)
4. N^C,0<C<1
5. N
6. N\*log(N)
7. N^C,C>1
8. C^N,C>1
9. N!

Если необходимо оценить сложность алгоритма, уравнение сложности которого содержит несколько этих функций, то уравнение можно сократить до функции, расположенной ниже в таблице. Например, O(log(N)+N!)=O(N!).

В общем случае сложность алгоритма можно оценить по порядку величины. Алгоритм имеет сложность O(f(n)), если при увеличении размерности входных данных N, время выполнения алгоритма возрастает с той же скоростью, что и функция f(N).

Оценивая порядок сложности алгоритма, необходимо использовать только ту часть, которая возрастает быстрее всего. Предположим, что рабочий цикл описывается выражением N^3+N. В таком случае его сложность будет равна O(N^3). Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении N. Например, при N=100, то разница между N^3+N=1000100 и N=1000000 равна всего лишь 100, что составляет 0,01%. При вычислении O можно не учитывать постоянные множители в выражениях. Алгоритм с рабочим шагом 3N^3 рассматривается, как O(N^3). Это делает зависимость отношения O(N) от изменения размера задачи более очевидной[[7](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

***Структуры данных***

Структура данных − это контейнер, который хранит данные в определенном макете. Этот «макет» позволяет структуре данных быть эффективной в некоторых операциях и неэффективной в других.

Линейные структуры данных, элементы образуют последовательность или линейный список, обход узлов линеен. Примеры: Массивы, связанный список, стеки и очереди. Нелинейные, если обход узлов нелинейный, а данные не последовательны. Пример: граф и деревья[[8](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)]

[Основные структуры данных:](https://habr.com/ru/post/422259/)

1. Линейный список[[3](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)] представляет собой последовательность n ≥ 0 узлов x[1], x[2], …, x[n], важнейшей структурной особенностью которой является такое расположение элементов списка один относительно другого, как будто они находятся на одной линии. Иначе говоря, в такой структуре должно соблюдаться следующее условие: если n > 0 и x[1] является первым узлом, а x[n] − последним, то k-й узел x[k] следует за x [k – 1] и предшествует узлу x[k + 1] для всех 1 < k < n.

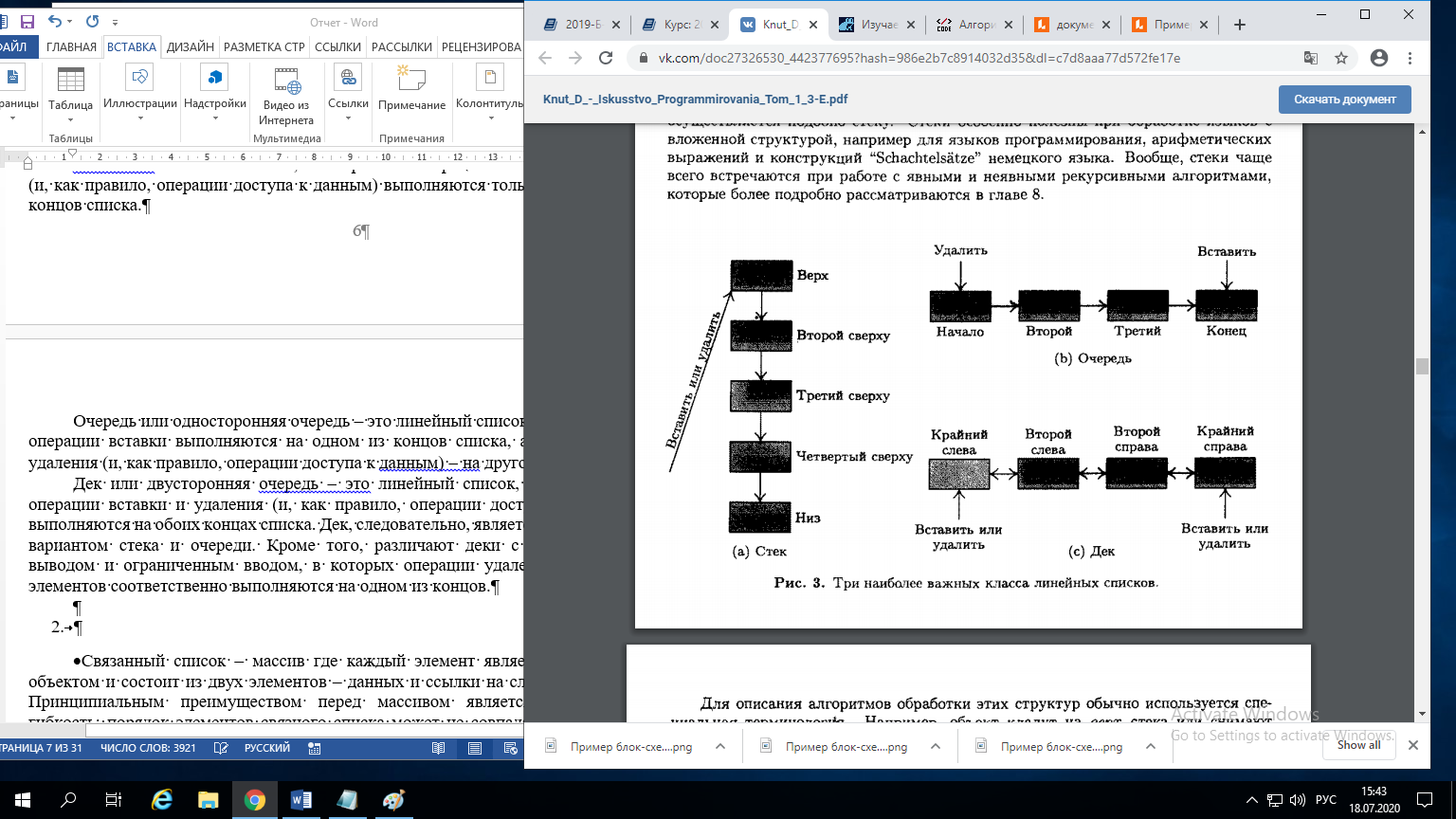
С линейными списками могут выполняться следующие операции:

* получение доступа к k-му элементу списка для проверки и/или изменения содержимого его полей;
* вставка нового узла сразу после или до k-го узла;
* удаление k-го узла;
* объединение в одном списке двух (или более) линейных списков;
* разбиение линейного списка на два (или более) списка;
* создание копии линейного списка;
* определение количества узлов в списке;
* сортировка узлов в порядке возрастания значений в определенных полях этих узлов;
* поиск узла с заданным значением в некотором поле.

Линейные списки, в которых операции вставки, удаления и доступа к значениям чаще всего выполняются в первом или последнем узле, получили следующие специальные названия.

Стек [ [рисунок 1 (а)](#_Рисунок_1_–) ] − это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка.

Очередь или односторонняя очередь [ [рисунок 1 (b)](#_Рисунок_1_–) ] – это линейный список, в котором все операции вставки выполняются на одном из концов списка, а все операции удаления (и, как правило, операции доступа к данным) − на другом.

Дек или двусторонняя очередь [ [рисунок 1 (c)](#_Рисунок_1_–) ] − это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются на обоих концах списка. Дек, следовательно, является более общим вариантом стека и очереди. Кроме того, различают деки с ограниченным выводом и ограниченным вводом, в которых операции удаления и вставки элементов соответственно выполняются на одном из концов.

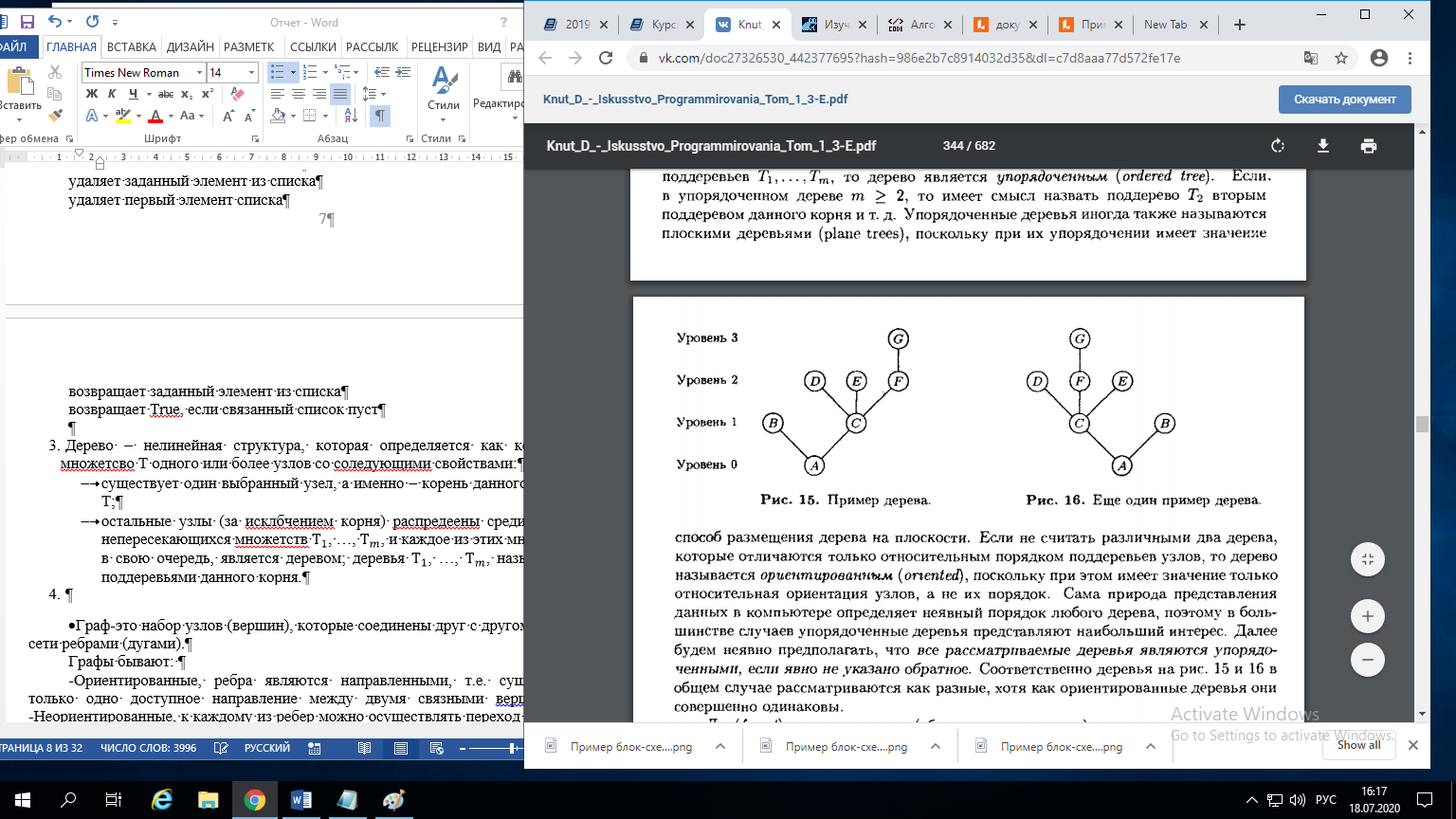
# Рисунок 1 – Три класса линейных списков [[3](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)]

1. Связанный список[[8](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)] – массив где каждый элемент является отдельным объектом и состоит из двух элементов – данных и ссылки на следующий узел.  
   Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.  
   Однонаправленный, каждый узел хранит адрес или ссылку на следующий узел в списке и последний узел имеет следующий адрес или ссылку как NULL.  
   1->2->3->4->NULL  
   Двунаправленный, две ссылки, связанные с каждым узлом, одним из опорных пунктов на следующий узел и один к предыдущему узлу.  
   NULL<-1<->2<->3->NULL  
   Круговой, все узлы соединяются, образуя круг. В конце нет NULL. 1->2->3->1

Основные операции:

* вставка заданного элемента в конец списка;
* вставка элемента в начало списка;
* удаляет заданный элемент из списка;
* удаляет первый элемент списка;
* возвращает заданный элемент из списка;
* возвращает True, если связанный список пуст;

1. Дерево [ [рисунок 2](#_Рисунок_2_–) ] – нелинейная структура, которая определяется как конечное множество Т одного или более узлов со следующими свойствами:

* существует один выбранный узел, а именно – корень данного дерева Т;
* остальные узлы (за исключением корня) распределены среди m ≥ 0 непересекающихся множеств , …, , и каждое из этих множеств, в свою очередь, является деревом; деревья , …, , называются поддеревьями данного корня.

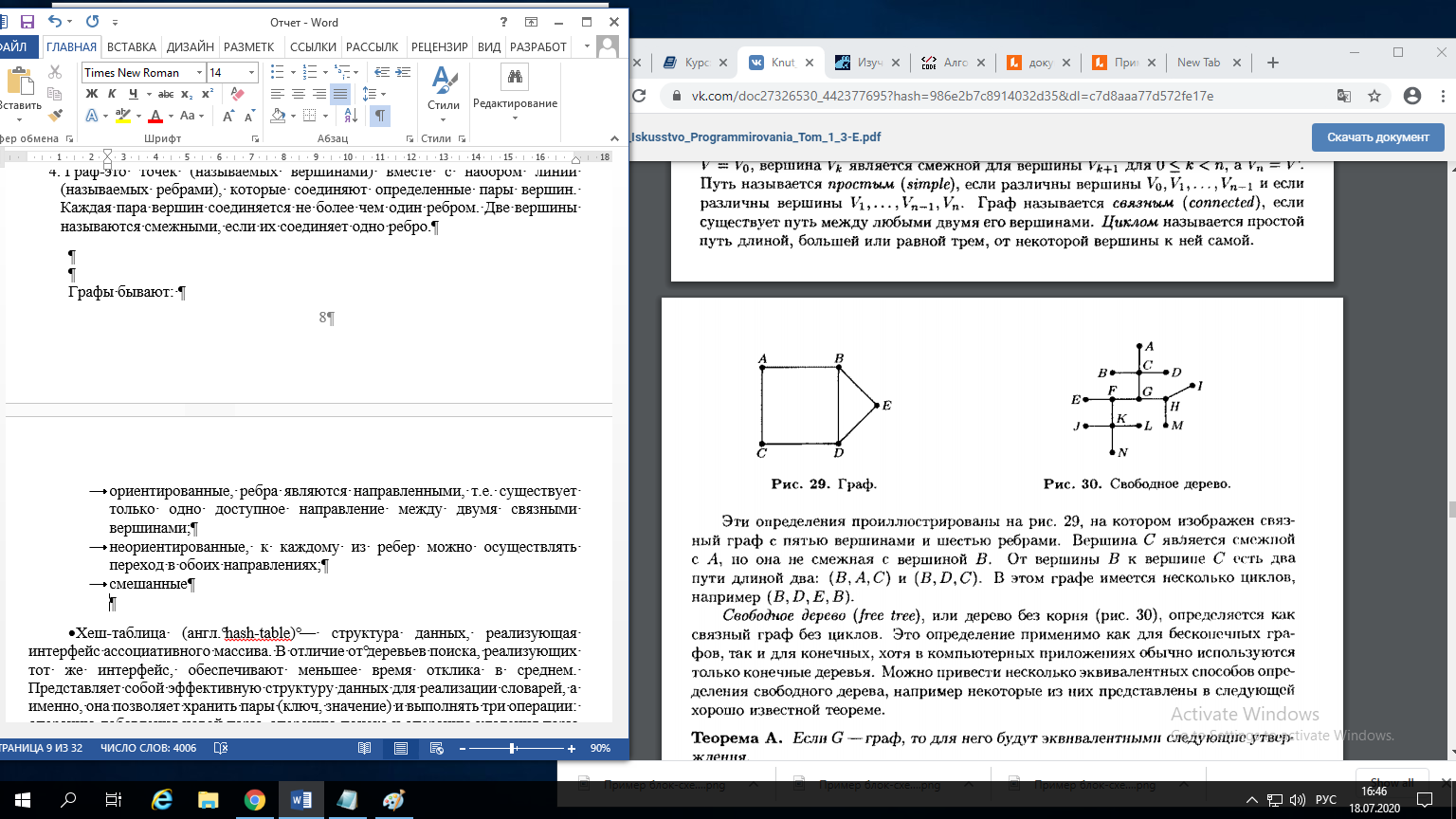
# Рисунок 2 – Пример дерева [[3](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)]

Бинарное дерево – конечное множество узлов, которое является пустым или состоит из корня и не более двух непересекающихся бинарных деревьев, которые называются левым и правым поддеревьями данного корня.

1. Граф [ [рисунок 3](#_Рисунок_3_–) ] − это множество точек (называемых вершинами) вместе с набором линий (называемых ребрами), которые соединяют определенные пары вершин. Каждая пара вершин соединяется не более чем один ребром. Две вершины называются смежными, если их соединяет одно ребро.

Графы бывают:

* ориентированные, ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами;
* неориентированные, к каждому из ребер можно осуществлять переход в обоих направлениях;
* cмешанные [[3](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

Рисунок 3 – Граф [[3](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)]

1. Хеш-таблица представляет из себя структуру данных, с помощью которой можно соотносить ключи со значениями. и выполнять три операции:

* добавление новой пары;
* поиск пары по ключу;
* удаление пары по ключу[[2](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

Как правило, хеш-таблицы реализуются при помощи массивов.  
Производительность зависит от следующих трех факторов:

* хеш-функция;
* размер хеш-таблицы;
* метод обработки коллизий.

Хеш-функция − функция, которая по каждому входящему элементу определяет натуральное число (хеш). Полученный хеш является индексом массива. Зная индекс, можно выполнить операции добавления, удаления или поиска за О(1).

Однако, когда хеш-функция выдает одинаковое натуральное число для разных элементов, возникает коллизия[[9](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)]

Количество коллизий зависит от хеш-функции; чем лучше используемая хеш-функция, тем меньше вероятность их возникновения.

Пока хеш-таблица не слишком заполонена, коллизии появляются редко и производительность схемы хеширования определяется прежде всего временем, требующимся для вычисления хеш-функции. По мере заполнения хеш-таблицы доступ к именам требует всё больше времени из-за коллизий. Поэтому эффективность схемы хеширования во многом определяется схемой разрешения коллизий[[5](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

Открытая адресация

Алгоритм вставки элемента проверяет ячейки массива в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Последовательность, в которой просматриваются ячейки хеш-таблицы, называется последовательностью проб.

Метод цепочек

Каждая ячейка хеш-таблицы является связным списком. При возникновении коллизии создается дополнительный элемент связного списка (который состоит из двух полей – поля данных и поля связки). В результате выстраивается цепочка из элементов списка, а в ячейке хеш-таблицы хранится указатель на первый элемент списка[[4](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

Основным достоинством методов хеширования является то, что соответствующее им среднее время поиска не зависит от размера таблицы, т.е. на практике они являются самыми быстрыми методами поиска.

В качестве недостатков методов хеширования можно отметить следующие:

* табличный порядок имен обычно не связан с их естественным порядком;
* худший случай может оказаться хуже, чем при последовательном поиске;
* сложность динамического расширения таблиц, поскольку расширение может приводить к потере памяти, если таблица слишком велика, или к малой производительности, если слишком мала[[5](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

# КРАТКОЕ ОПИСАНИЕ АЛГОРИТМА ЗАДАНИЕ А

[Сортировка простыми обменами, сортировка пузырьком](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC#.D0.9E.D0.BF.D1.82.D0.B8.D0.BC.D0.B8.D0.B7.D0.B0.D1.86.D0.B8.D1.8F) [ [рисунок 4](#_Рисунок_4_–) ] – один из квадратичных алгоритмов сортировки.

Сортировка пузырьком использует два цикла. Во вложенном цикле выполняется попарное сравнение элементов и, в случае нарушения порядка их следования, перестановка. В результате выполнения одной итерации внутреннего цикла, максимальный элемент гарантированно будет смещен в конец массива. Внешний цикл выполняется до тех пор, пока весь массив не будет отсортирован [[6](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

Оптимизация:

* можно заметить, что после i-ой итерации внешнего цикла i последних элементов уже находятся на своих местах в отсортированном порядке, поэтому нет необходимости производить их сравнения друг с другом. Следовательно, внутренний цикл можно выполнять не до n−2, а до n−i−2;
* если после выполнения внутреннего цикла не произошло ни одного обмена, то массив уже отсортирован, и продолжать что-то делать бессмысленно. Поэтому внутренний цикл можно выполнять не n−1 раз, а до тех пор, пока во внутреннем цикле происходят обмены.

[Сортировка вставками](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8) [ [рисунок 5](#_Рисунок_5_–) ] – квадратичный алгоритм сортировки.

На каждом шаге алгоритма выбирается первый элемент необработанной части массива и вставляется в отсортированную так, чтобы в ней сохранялся требуемый порядок следования элементов. Вставка может выполняться как в конец массива, так и в середину. При вставке в середину необходимо сдвинуть все элементы, расположенные «правее» позиции вставки на один элемент вправо. В алгоритме используется два цикла − в первом выбираются элементы необработанной части, а во втором осуществляется вставка[[6](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

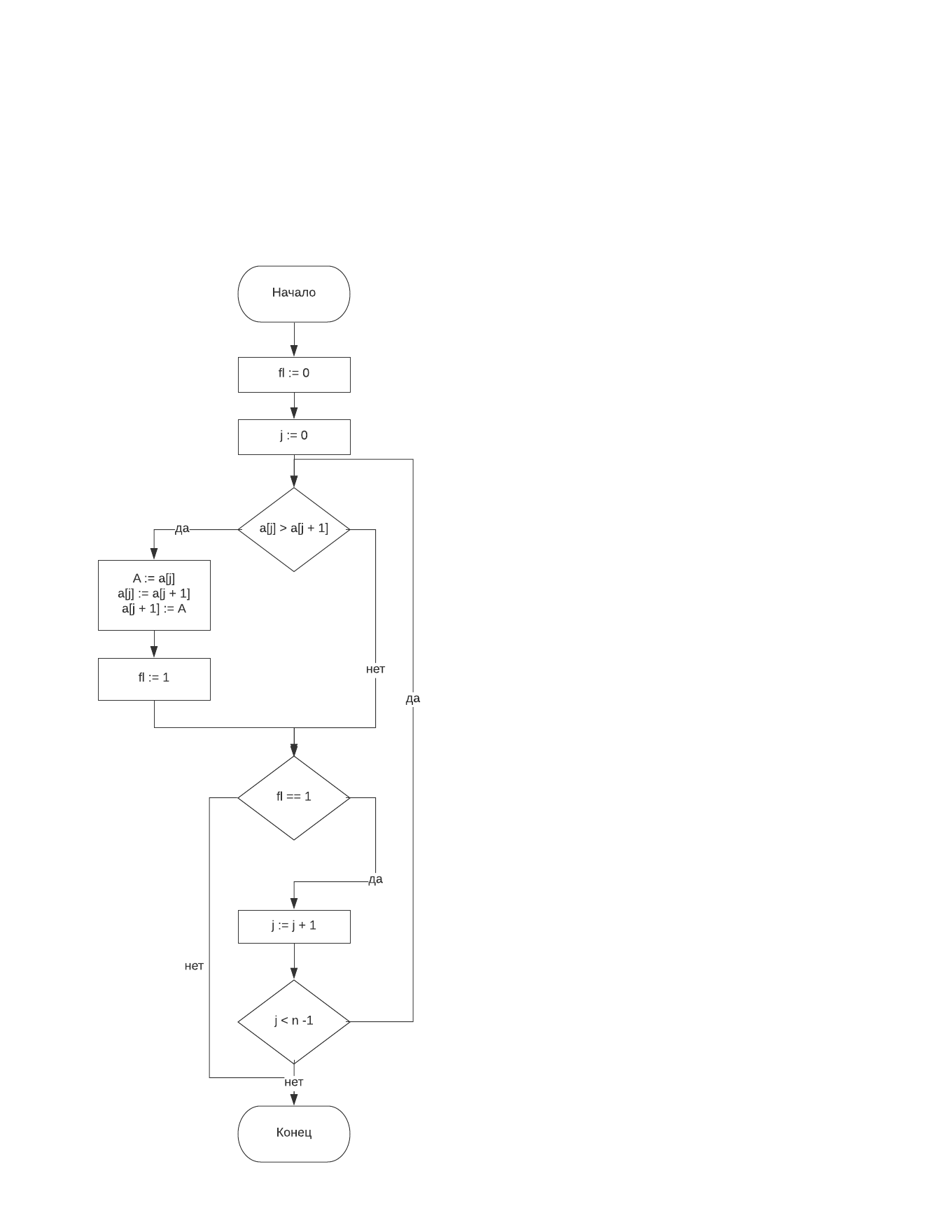
Алгоритм работает в среднем и в худшем случае – за O(n^2). Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие – когда они расположены в обратном порядке.

[Сортировка выбором](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%8B%D0%B1%D0%BE%D1%80%D0%BE%D0%BC) [ [рисунок 6](#_Рисунок_6_–) ] – простой алгоритм сортировки со сложностью O(n^2), где n − количество элементов для сортировки

В сортировке выбором массив разделяется на отсортированную и необработанную части. Изначально отсортированная часть пустая, но постепенно она увеличивается. Алгоритм производит поиск минимального элемента необработанной части и меняет его местами с первым элементом той же части, после чего считается, что первый элемент обработан (отсортированная часть увеличивается) [[6](#_БИБЛИОГРАФИЧЕСКИЙ_СПИСОК)].

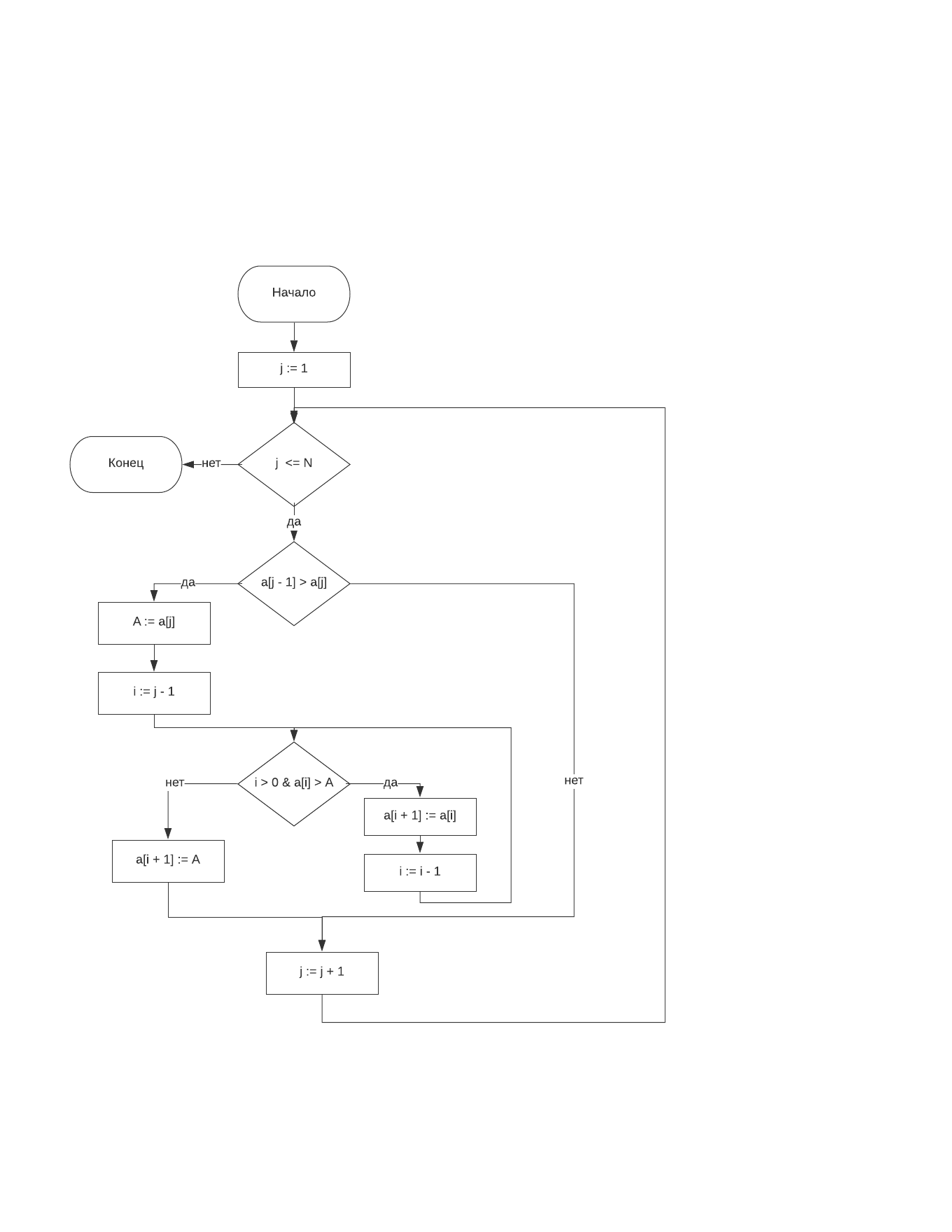
# СХЕМА АЛГОРИТМА ЗАДАНИЯ А

**Сортировка пузырьком**



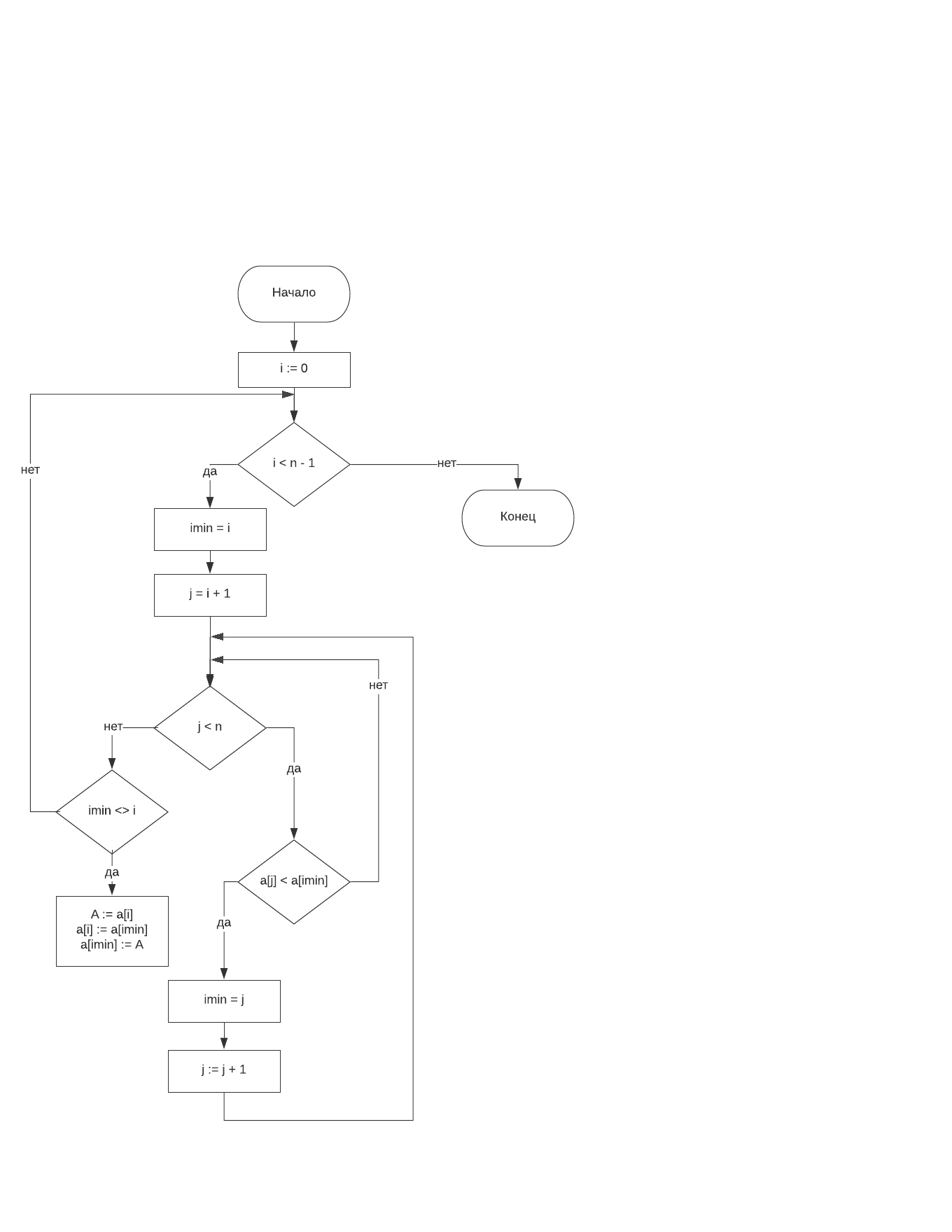
# Рисунок 4 – Блок-схема сортировки пузырьком

**Сортировка** вставками



# Рисунок 5 – Блок-схема сортировки вставками

**Сортировка выбором**



# Рисунок 6 – Блок-схема сортировки выбором

# ЭКСПЕРИМЕНТАЛЬНЫЙ ГРАФИК ЗАВИСИМОСТИ ВРЕМЕНИ ВЫПОЛНЕНИЯ КОДА

Сортировка пузырьком[ [рисунок 7](#_Рисунок_7_–_1) ]

# Рисунок 7 – График. Сортировка пузырьком. Экспериментальные данные из [таблицы 1](#_Таблица_1_-)

# Таблица 1 - Сортировка пузырьком. Экспериментальные данные

|  |  |  |  |
| --- | --- | --- | --- |
|  | Случайная последовательность | Последовательность, отсортированная в обратном порядке | Отсортированная последовательность |
| 10 000 | 1.123 | 1.305 | 0.000 |
| 50 000 | 23.796 | 29.950 | 0.000 |
| 100 000 | 103.370 | 125.423 | 0.000 |
| 250 000 | 868.254 | 989.402 | 0.000 |
| 500 000 | 2673.927 | 3485.297 | 0.007 |

Сортировка вставками[ [рисунок 8](#_Рисунок_8_–_1) ]

# Рисунок 8 – График. Сортировка вставками. Экспериментальные данные из [таблицы 2](#_Таблица_2_-)

# Таблица 2 - Сортировка вставками. Экспериментальные данные

|  |  |  |  |
| --- | --- | --- | --- |
|  | Случайная последовательность | Последовательность, отсортированная в обратном порядке | Отсортированная последовательность |
| 10 000 | 0.238 | 0.473 | 0.000 |
| 50 000 | 6.924 | 13.464 | 0.001 |
| 100 000 | 22.408 | 47.193 | 0.001 |
| 250 000 | 137.050 | 305.308 | 0.002 |
| 500 000 | 718.666 | 1405.637 | 0.009 |

Сортировка выбором[ [рисунок 9](#_Рисунок_9_–_1) ]

# Рисунок 9 – График. Сортировка выбором. Экспериментальные данные из [таблицы 3](#_Таблица_3_-)

# Таблица 3 − Сортировка выбором. Экспериментальные данные.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Случайная последовательность | Последовательность, отсортированная в обратном порядке | Отсортированная последовательность |
| 10 000 | 0.346 | 0.245 | 0.233 |
| 50 000 | 13.731 | 10.845 | 8.606 |
| 100 000 | 41.097 | 30.941 | 27.759 |
| 250 000 | 147.499 | 169.712 | 164.884 |
| 500 000 | 720.693 | 769.371 | 760.064 |

# СОПОСТАВЛЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ С ТЕОРЕТИЧЕСКИМИ СЛОЖНОСТЯМИ АЛГОРИТМОВ (ПО ВРЕМЕНИ ИСПОЛНЕНИЯ)

Алгоритмы сортировки пузырьком, вставками и выбором относятся к самым простым и легко реализуемым. Тем не менее они считаются одними из наименее эффективных. Их теоретическая сложность – О([ [рисунок 10](#_Рисунок_10_–) ]. Стоит отметить, что время сортировки не будет равно квадрату размера массива данных, так как скорость сортировки зависит не только от алгоритма, но и от технических характеристик ПК или ноутбука, а также от других программ и приложений, которые могут работать параллельно с сортировкой массива.

# Рисунок 10 – График. Теоретическая сложность алгоритмов О()

Графики [ [рисунок 11](#_Рисунок_11_-) ] , построенные по значениям, которые были получены экспериментальным путем, соответствуют графику теоретической сложности алгоритмов, следовательно, можно сделать вывод, что экспериментальные и теоретические оценки работы алгоритмов совпадают и равны О(

# Рисунок 11 - График. Сравнение экспериментальных данных работы алгоритмов с теоретической оценкой сложности О ()

Если рассматривать массивы одинаковые по количеству элементов, но разные по модулю значений (10; 100; ; ), то скорость работы алгоритмов от этого существенно не меняется. Следовательно, данные алгоритмы не зависят от величины сортируемых элементов [ [рисунок 12](#_Рисунок_12_–) ].

rand () %10

rand () %100

rand () %1000

rand () %1е5

# Рисунок 12 – График. Зависимость времени работы алгоритмов от модуля элементов массива.

В результате проведенного исследования и полученных данных, для сортировки наиболее оптимальной из представленных является сортировка вставками. Пузырьковая сортировка в сравнении с другими работает очень медленно, исключением является лишь уже отсортированный массив. Сортировка выбором и вставками работают почти одинаково по времени, но сортировка вставками не относится к устойчивым алгоритмам сортировки (устойчивость − устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами), следовательно, она работает с уже отсортированным массивом также долго, как и со случайной последовательностью.

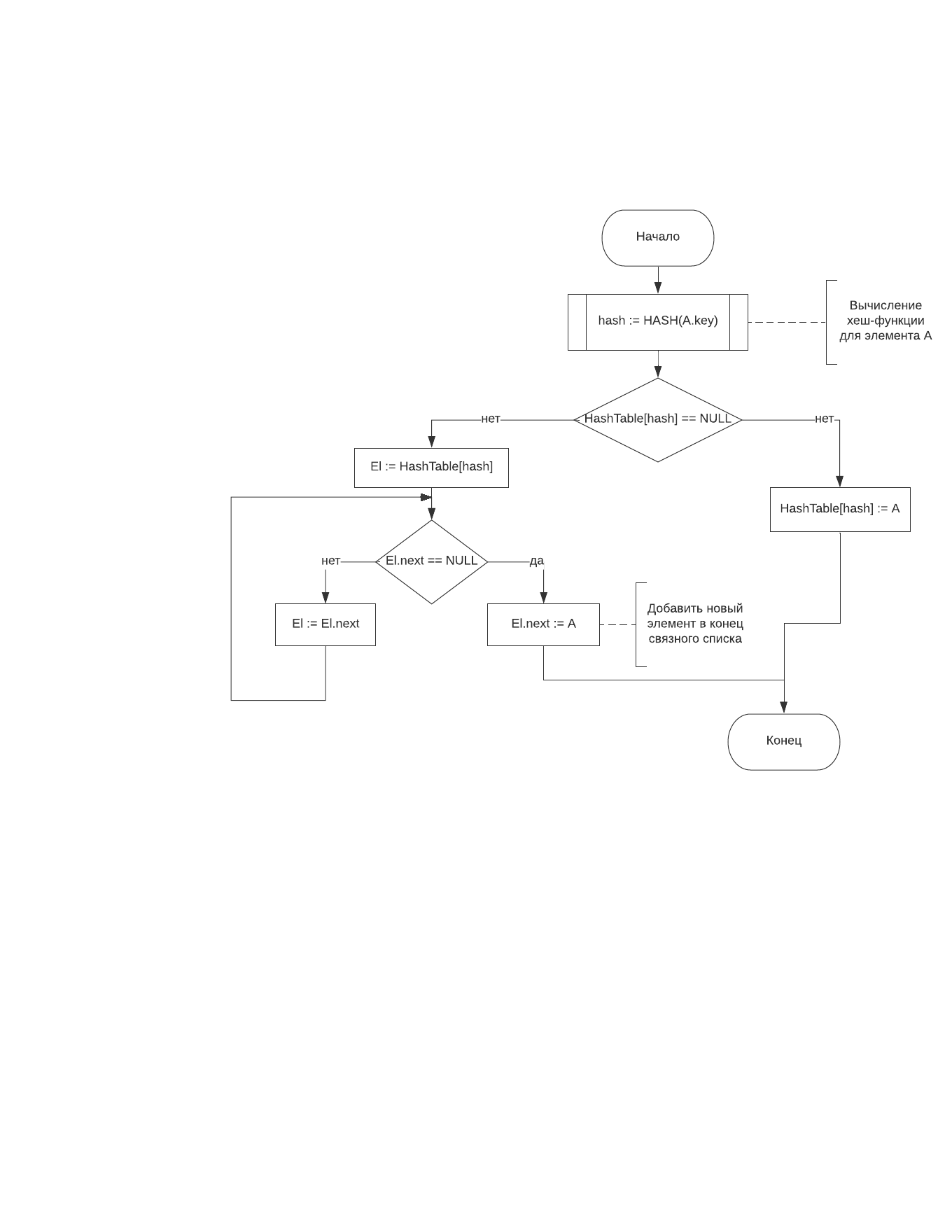
# КРАТКОЕ ОПИСАНИЕ АЛГОРИТМА ЗАДАНИЯ В

1. Необходимо реализовать ассоциативный массив, в котором доступ до элемента О(1). Для этого идеально подходит хеш-таблица, которая является обычным массивом с необычной адресацией, задаваемой хеш-функцией. Чем сложнее и продуманней хеш-функция, тем меньше вероятность возникновения коллизий, то есть равенства хеш-значений для двух различных ключей.
2. Коллизий в любом случае не избежать, поэтому для решения этой проблемы отлично подходит метод цепочек. Данный метод заключается в том, что каждая ячейка массива содержит не один элемент, а односвязный список. Следовательно, при возникновении коллизии элемент помещается в соответствующую ячейка в конец односвязного списка. Тогда доступ до элемента в самом худшем случае будет равен О(n) и будет столь же эффективен как и обычный массив, тем не менее в среднем хеш-таблица будет достаточно эффективна и её работа будет оцениваться как О(1).
3. Сообщения об ошибке выводятся в двух случаях:

* входной текстовый файл не найден (программа не может работать)
* считанная команда не существует (происходит переход к следующей команде)

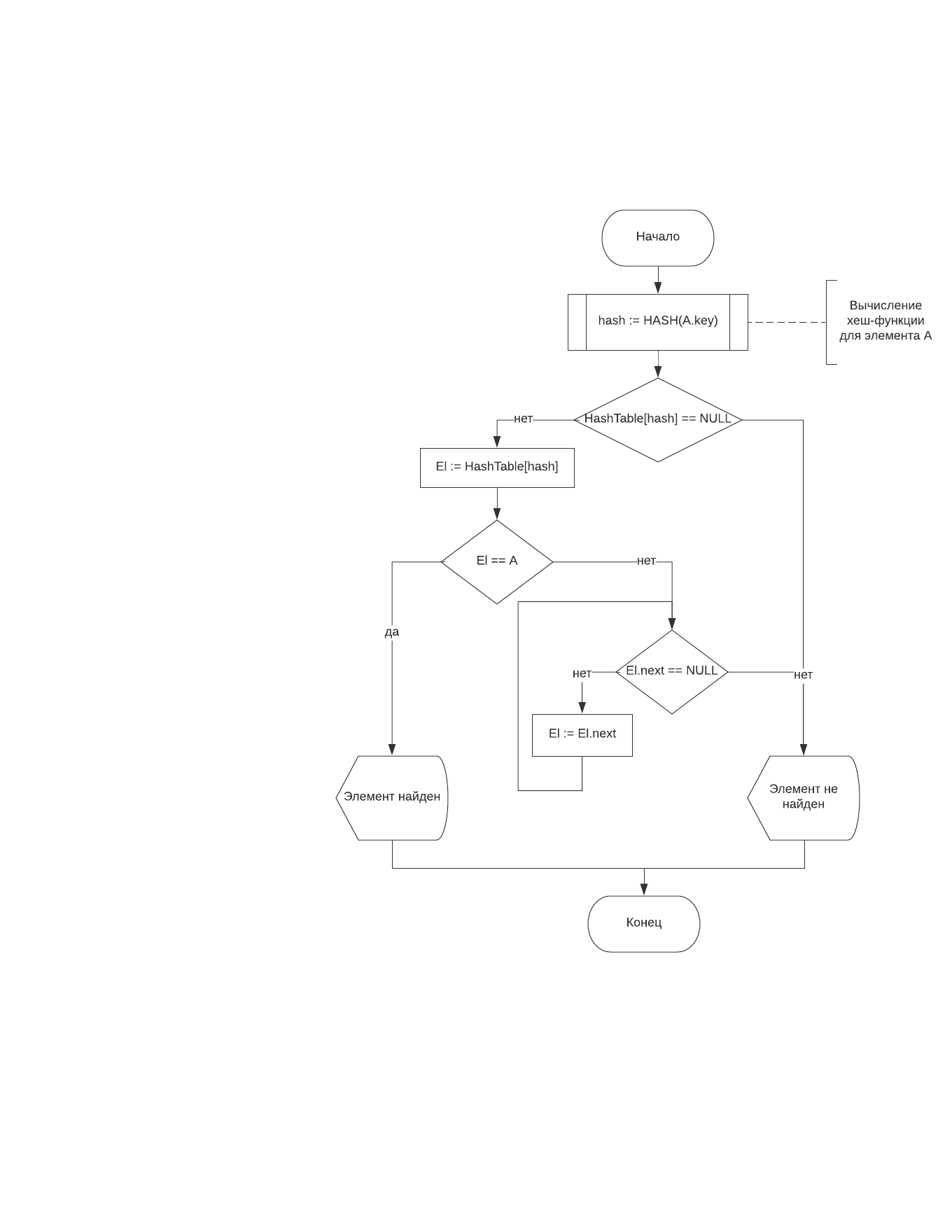
# СХЕМА АЛГОРИТМА ЗАДАНИЯ В

Функция INSERT



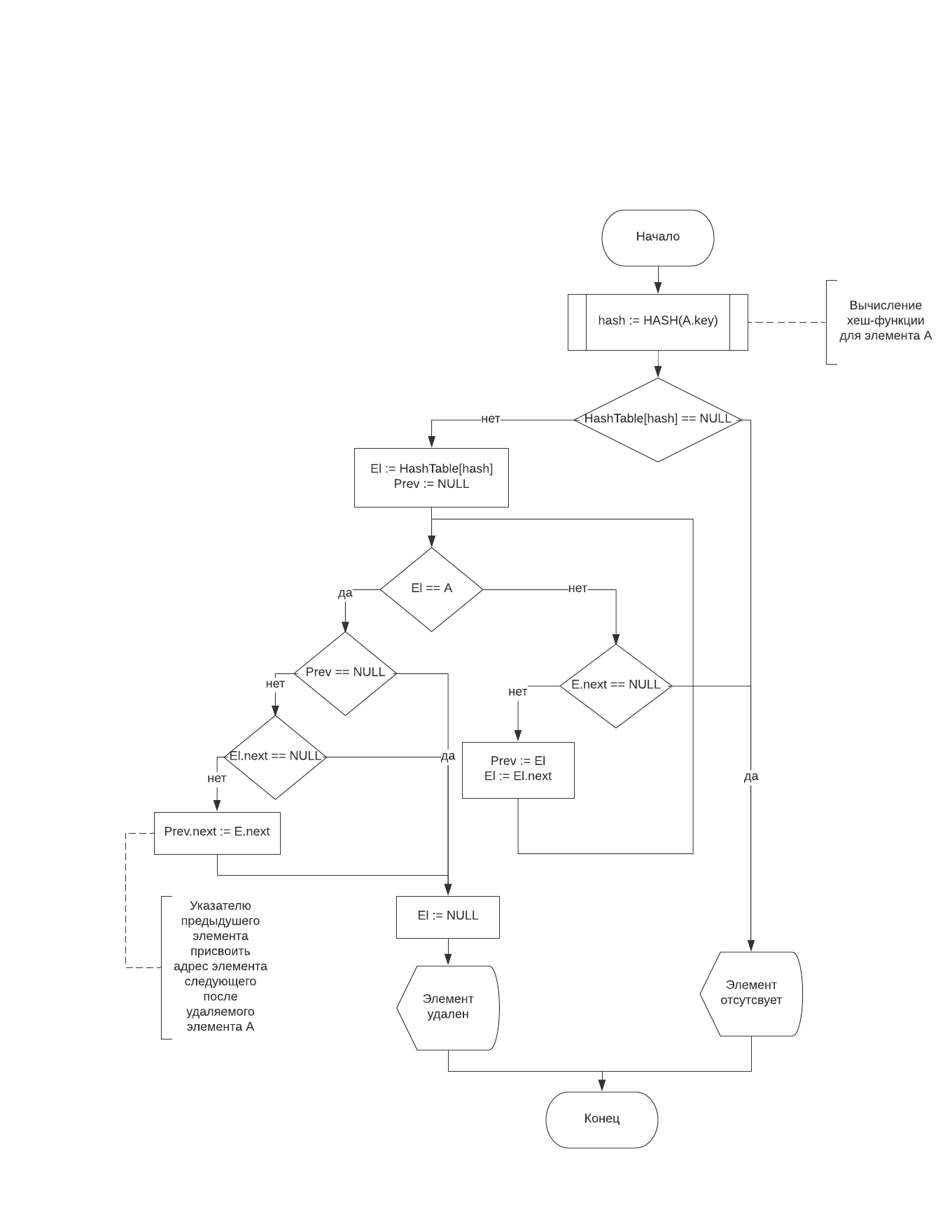
# Рисунок 13 – Блок-схема функции добавления элемента в хеш-таблицу

Функция FIND



# Рисунок 14 – Блок-схема функции поиска элемента в хеш-таблице

Функция REMOVE



# Рисунок 15 – Блок-схема функции удаления элемента из хеш-таблицы

# ОПИСАНИЕ РЕАЛИЗАЦИИ ЗАДАНИЯ В

1. Всякий доступ к таблице должен быть произведен через хэш-функцию. Функция должна порождать хорошие ключи для распределения элементов по таблице, что в свою очередь минимизирует коллизии (случаи, когда два разных элемента имеют одинаковое значение хеш-функции) и предотвратит случай, когда элементы данных с близкими значениями попадают только в одну часть таблицы.

Так как ключи – это строки, то необходимо их преобразовать в число, чтобы определить индекс массива, в который будет помещен элемент. Для нахождения индекса суммируются коды букв ключа (таблица [ASCII](https://ru.wikipedia.org/wiki/ASCII)) и берется остаток от деления на размер хеш-таблицы.

1. При добавлении элементов в хеш-таблицу выделяются динамическая память, которая организуется в виде односвязных списков, каждый из которых соответствует входу хеш-таблицы. Этот метод называется связыванием, открытой адресацией или методом цепочек. Таким образом, достаточно эффективно разрешаются коллизии.
2. Чтобы вставить в таблицу новый элемент, хешируется его ключ, для определения списка, в который нужно добавить элемент. Затем элемент вставляется в начало этого списка[ рисунок 13 ]. Чтобы найти элемент, хешируется его ключ и осуществляется проход по соответствующему списку [рисунок 14 ]. Чтобы удалить элемент, хешируется ключ, осуществляется проход по соответствующему списку (также необходимо запомнить предыдущий элемент, если он имеется). Если элемент удалось найти, необходимо переопределить указатели (для этого и нужен предыдущий элемент, если он имеется) и удалить элемент из списка [рисунок 15 ].
3. Если встречается недопустимая команда, то выводится соответствующее сообщение на консоль ("!Invalid command!"). Цикл while позволяет перейти к следующей команде пока не встретиться команда END, которая завершает считывание команд из файла.
4. Во входном файле команды записаны построчно в виде INSERT(“ключ”, значение), последняя команда END.
5. При добавлении элемента в хеш-таблицу выводится сообщение на консоль «The element is inserted». При удалении из таблицы – «The element is removed!» (если он был в таблице) или «ERROR.The element does not exist!» (если элемент отсутствует). При выполнении поиска элемента либо выводится значение, которое принадлежит данному ключу и сообщение «The element is found!», либо «ERROR.The element does not exist!» при отсутсвии элемента.

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Абрамов С.А. Лекции о сложности алгоритмов – М.: МЦНМО, 2009. – 256 с.
2. Башир И. Блокчейн: архитектура, криптовалюты, инструменты разработки, смарт-контакты / пер. с анг. М. А. Райтмана. – М.: ДМК Пресс, 2019. – 538 с.
3. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы / Д. Кнут. - М.: Диалектика, 2019. - 720 c.
4. Осипов Д.Л. Технологии проектирования баз данных. М.: ДМК Пресс, 2019. – 498 с.
5. Павлов Л. А. Структуры и алгоритмы обработки данных: учебник / Л. А. Павлов, Н. В. Пернова – 2-е изд., испр. и доп. – Санкт – Петербург : Лань, 2020. – 256 с.
6. Блог программиста. Блок-схемы алгоритмов. ГОСТ. Примеры − <https://pro-prof.com/archives/1462> Дата обращения: 18.06.2020
7. Хабр. Оценка сложности алгоритмов − <https://habr.com/ru/post/104219/> Дата обращения: 18.06.2020
8. Хабр. Основные структуры данных. Матчасть. Азы − <https://habr.com/ru/post/422259/> Дата обращения: 11.06.2020
9. Хабр. Хеш-таблицы − <https://habr.com/ru/post/509220/> Дата обращения: 18.06.2020

# ПРИЛОЖЕНИЕ А

**ИСХОДНЫЙ КОД задание А**

*Сортировка пузырьком*

#include <stdio.h>

#include <malloc.h>

#include <time.h>

#include <stdlib.h>

void BubbleSort(int \*array, int N)

{

int j, fl;

do

{

fl = 0;

for (j = 0; j < (N - 1); j++)

{

if (array[j] > array[j + 1])

{

int tmp = array[j];

array[j] = array[j + 1];

array[j + 1] = tmp;

fl = 1;

}

}

} while(fl);

}

int main()

{

FILE \*input;

if ((input = fopen("input.txt", "r")) == NULL)

{

printf("!ERROR!File not found!\n");

return 0;

}

int \*arrayU, \*arrayR, \*arrayS, N, i;

double t1, t2, t3;

fscanf(input, "%i", &N);

printf("%i", N);

arrayU = (int\*)malloc(N \* sizeof(int));

arrayR = (int\*)malloc(N \* sizeof(int));

arrayS = (int\*)malloc(N \* sizeof(int));

for (i=0; i < N; i++)

{

arrayU[i] = rand()%100000;

//arrayR[i] = N - i;

//arrayS[i] = i;

}

t1 = clock();

BubbleSort(arrayU, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nUnsorted array\nTIME: %f", t3);

t1 = clock();

BubbleSort(arrayR, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nReverse sorted array\nTIME: %f", t3);

t1 = clock();

BubbleSort(arrayS, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nSorted array\nTIME: %f", t3);

free(arrayU);

free(arrayR);

free(arrayS);

fclose(input);

return 0;

}

*Сортировка вставками*

#include <stdio.h>

#include <malloc.h>

#include <time.h>

#include <stdlib.h>

void InclusionSort(int \*array, int N)

{

int i, index, value;

for (i = 1; i < N; i++)

{

index = i;

value = array[i];

while ((index > 0) && (array[index - 1] > value))

{

array[index] = array[index - 1];

index--;

}

array[index] = value;

}

}

int main()

{

FILE \*input;

if ((input = fopen("input.txt", "r")) == NULL)

{

printf("!ERROR!File not found!\n");

return 0;

}

int \*arrayU, \*arrayR, \*arrayS, N, i;

double t1, t2, t3;

fscanf(input, "%i", &N);

printf("%i", N);

arrayU = (int\*)malloc(N \* sizeof(int));

arrayR = (int\*)malloc(N \* sizeof(int));

arrayS = (int\*)malloc(N \* sizeof(int));

for (i=0; i < N; i++)

{

arrayU[i] = rand()%100000;

arrayR[i] = N - i;

arrayS[i] = i;

}

t1 = clock();

InclusionSort(arrayU, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nUnsorted array\nTIME: %f", t3);

t1 = clock();

InclusionSort(arrayR, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nReverse sorted array\nTIME: %f", t3);

t1 = clock();

InclusionSort(arrayS, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nSorted array\nTIME: %f", t3);

free(arrayU);

free(arrayR);

free(arrayS);

fclose(input);

return 0;

}

*Сортировка выбором*

#include <stdio.h>

#include <malloc.h>

#include <time.h>

#include <stdlib.h>

void SelectionSort(int \*array, int N)

{

int i, j, min, tmp;

for (i = 0; i < (N - 1); i++)

{

min = i;

for (j = (i + 1); j < N; j++)

{

if (array[min] > array[j])

{

min = j;

}

}

tmp = array[min];

array[min] = array[i];

array[i] = tmp;

}

}

int main()

{

FILE \*input;

if ((input = fopen("input.txt", "r")) == NULL)

{

printf("!ERROR!File not found!\n");

return 0;

}

int \*arrayU, \*arrayR, \*arrayS, N, i;

double t1, t2, t3;

fscanf(input, "%i", &N);

printf("%i", N);

arrayU = (int\*)malloc(N \* sizeof(int));

arrayR = (int\*)malloc(N \* sizeof(int));

arrayS = (int\*)malloc(N \* sizeof(int));

for (i=0; i < N; i++)

{

arrayU[i] = rand()%100000;

arrayR[i] = N - i;

arrayS[i] = i;

}

t1 = clock();

SelectionSort(arrayU, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nUnsorted array\nTIME: %f", t3);

t1 = clock();

SelectionSort(arrayR, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nReverse sorted array\nTIME: %f", t3);

t1 = clock();

SelectionSort(arrayS, N);

t2 = clock();

t3 = (t2-t1) / CLOCKS\_PER\_SEC;

printf("\nSorted array\nTIME: %f", t3);

free(arrayU);

free(arrayR);

free(arrayS);

fclose(input);

return 0;

}

# ПРИЛОЖЕНИЕ В

**ИСХОДНЫЙ КОД задание В**

*Main.c*

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <locale.h>

#include <string.h>

#include "lib.h"

struct elem {

char \*key;

int value;

struct elem\* next;

};

int main() {

int value, fl = 1, i, d;

char key[15], buff[100];;

struct elem \*el;

FILE\* input;

if ((input = fopen("input.txt", "r")) == NULL)

{

printf("!ERROR!File not found!\n");

return 0;

}

printf("Associative array");

while (fl){

fscanf(input, "%s", &buff);

printf("\n\n%s", buff);

switch(buff[0]) {

case 'E':

fl = 0;

break;

case 'I':

memset(&key, '\0', sizeof(key));

for(i = 8; buff[i] != '"'; i++) key[i-8] = buff[i];

value = 0; d = 1;

for(i = strlen(buff) - 2; buff[i] != ','; i--)

{

value += (buff[i]-'0') \* d;

d \*= 10;

}

if ((el =INSERT(key, value)) == NULL) printf("\nERROR!");

else

{

printf ("\nKey: %s", el->key);

printf("\nValue: %i", el->value);

printf("\nThe element is inserted.");

}

break;

case 'F':

memset(&key, '\0', sizeof(key));

for(i = 6; buff[i] != '"'; i++) key[i-6] = buff[i];

if ((el =FIND(key)) == NULL) printf("\nERROR.The element does not exist!");

else

{

printf("\nValue: %i", el->value);

printf("\nThe element is found!");

}

break;

case 'R':

memset(&key, '\0', sizeof(key));

for(i = 8; buff[i] != '"'; i++) key[i-8] = buff[i];

REMOVE(key);

break;

default:

printf("\n!Invalid command!");

}

}

fclose(input);

return 0;

}

*lib.h*

#ifndef LIB\_H

#define LIB\_H

struct elem \*FIND(char \*key);

struct elem \*INSERT(char \*key, int value);

void REMOVE(char \*key);

#endif //LIB\_H

*lib.c*

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <locale.h>

#include <string.h>

#include "lib.h"

static struct elem \*array[100];

struct elem {

char \*key;

int value;

struct elem\* next;

};

unsigned HASH(char \*key)

{

unsigned int hash = 0, j;

for (j = 0; j < strlen(key); j++) {

hash += (int)key[j];

}

return (hash % 100);

}

struct elem \*FIND(char \*key)

{

struct elem \*el;

for (el = array[HASH(key)]; el != NULL; el = el->next)

if (strcmp(key, el->key) == 0)

return el;

return NULL;

}

struct elem \*INSERT(char \*key, int value)

{

struct elem \*el;

unsigned hash;

if ((el = (FIND(key))) == NULL)

{

el = (struct elem \*)malloc(sizeof(\*el));

if (el == NULL || (el->key = strdup(key)) == NULL)

return NULL;

hash = HASH(key);

el->value = value;

el->next = array[hash];

array[hash] = el;

} else el->value = value;

return el;

}

void REMOVE(char \*key)

{

struct elem \*el, \*prev;

unsigned hash;

prev = 0;

hash = HASH(key);

el = array[hash];

while (el && (strcmp(el->key, key)!= 0))

{

prev = el;

el = el->next;

}

if (el == NULL)

{

printf("\nERROR.The element does not exist!");

return;

}

if (prev)

prev->next = el->next;

else

array[hash] = el->next;

printf("\nThe element is removed!");

free(el);

}