Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №7 по курсу "Анализ алгоритмов"

Тема Поиск в словаре

Студент Пересторонин П.Г.

Группа ИУ7-53Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

Оглавление

Введение

Словарь, как тип данных, применяется везде, где есть связь “ключ - значение” или “объект - данные”: поиск истории болезни пациента по номеру его амбулаторной карты, поиск налогов по ИНН и другое. Поиск - основная задача при использовании словаря. Данная задача решается различными способами, которые дают различную скорость решения.

Цель данной работы: получить навык работы со словарём, как структурой данных, реализовать алгоритмы поиска по словарю с применением оптимизаций.

В рамках выполнения работы необходимо решить следующие задачи:

- реализовать алгоритм поиска по словарю, использующий полный перебор;
- реализовать алгоритм поиска по словарю, использующий двоичный поиск;
- применить частотный анализ для эффективного поиска по словарю;
- сравнить полученные результаты;
- сделать выводы по проделанной работе.

1 Аналитическая часть

Словарь (или "*ассоциативный массив*")[1] - абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- `ВСТАВКА(ключ, значение)`;
- `ПОИСК(ключ)`;
- `УДАЛЕНИЕ(ключ)`.

В паре `(k, v)` значение `v` называется значением, ассоциированным с ключом `k`. Где `k` — это ключ, а `v` — значение. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Операция `ПОИСК(ключ)` возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект `НЕ_НАЙДЕНО`, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением, возможно, информации о том, успешно ли была выполнена данная операция).

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов — например, строки.

1.1 Алгоритм полного перебора

Алгоритмом полного перебора [2] называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты. В нашем случае мы последовательно будем перебирать ключи словаря до тех пор, пока не найдём нужный. Трудоёмкость алгоритма зависит от того, присутствует ли искомый ключ в словаре, и, если присутствует - насколько он далеко от начала массива ключей.

Пусть алгоритм нашёл элемент на первом сравнении (лучший случай), тогда будет затрачено $k_0 + k_1$ операций, на втором - $k_0 + 2 \cdot k_1$, на последнем (худший случай) - $k_0 + N \cdot k_1$. Если ключа нет в массиве ключей, то мы сможем понять это, только перебрав все ключи, таким образом трудоёмкость такого случая равно трудоёмкости случая с ключом на последней позиции. Средняя трудоёмкость может быть рассчитана как математическое ожидание по формуле (1.1), где Ω – множество всех возможных случаев.

$$\begin{aligned}\sum_{i \in \Omega} p_i \cdot f_i =& (k_0 + k_1) \cdot \frac{1}{N+1} + (k_0 + 2 \cdot k_1) \cdot \frac{1}{N+1} + \\ &+ (k_0 + 3 \cdot k_1) \cdot \frac{1}{N+1} + (k_0 + Nk_1) \frac{1}{N+1} + (k_0 + N \cdot k_1) \cdot \frac{1}{N+1} = \\ &= k_0 \frac{N+1}{N+1} + k_1 + \frac{1 + 2 + \cdots + N + N}{N+1} = \\ &= k_0 + k_1 \cdot \left(\frac{N}{N+1} + \frac{N}{2} \right) = k_0 + k_1 \cdot \left(1 + \frac{N}{2} - \frac{1}{N+1} \right)\end{aligned} \tag{1.1}$$

1.2 Алгоритм поиска в упорядоченном словаре двоичным поиском

При двоичном поиске [3] обход можно представить деревом, поэтому трудоёмкость в худшем случае составит $\log_2 N$ (в худшем случае нужно спуститься по двоичному дереву от корня до листа). Скорость роста функции $\log_2 N$ меньше, чем скорость линейной функции, полученной для полного перебора.

1.3 Частотный анализ

Алгоритм на вход получает словарь и на его основе составляется частотный анализ. По полученным значениям словарь разбивается на сегменты так, что все элементы с некоторым общим признаком попадают в один сегмент (для букв это может быть первая буква, для чисел - остаток

от деления).

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементам с наибольшей частотной характеристикой был самый быстрый доступ. Такой характеристикой может послужить, например, размер сегмента. Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам, то есть $P_i = \sum_j p_j = N \cdot p$, где P_i - вероятность обращения к i-ому сегменту, p_j - вероятность обращения к j-ому элементу, который принадлежит i-ому сегменту. Если обращения ко всем ключам равновероятны, то можно заменить сумму на произведение, где N - количество элементов в i-ом сегменте, а p - вероятность обращения к произвольному ключу.

Далее ключи в каждом сегменте упорядочиваются по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск со сложностью $O(\log_2 m)$ (где m - количество ключей в сегменте) внутри сегмента.

Таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента. Средняя трудоёмкость при множестве всех возможных случаев Ω может быть рассчитана по формуле (1.2).

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента i-ого элемента}} + f_{\text{бинарный поиск i-ого элемента}}) \cdot p_i \qquad (1.2)$$

Вывод

В данном разделе был рассмотрен абстрактный тип данных словарь и возможные реализации поиска в нём.

2 Конструкторская часть

2.1 Структура словаря

Словарь состоит из пар вида `<id - ФИ>`, где `id` - id игрока NHL [4] на оффициальном сайте лиги [4], `ФИ` - его фамилия и имя.

2.2 Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма поиска полным перебором, на рисунке 2.2 представлена схема поиска с использованием двоичного поиска, на рисунках 2.3 и 2.4 представлена схема поиска по сегментам, которые в результате частотного анализа (анализа длины) упорядочены в порядке убывания длины и отсортированы для возможности использования двоичного поиска внутри сегмента.

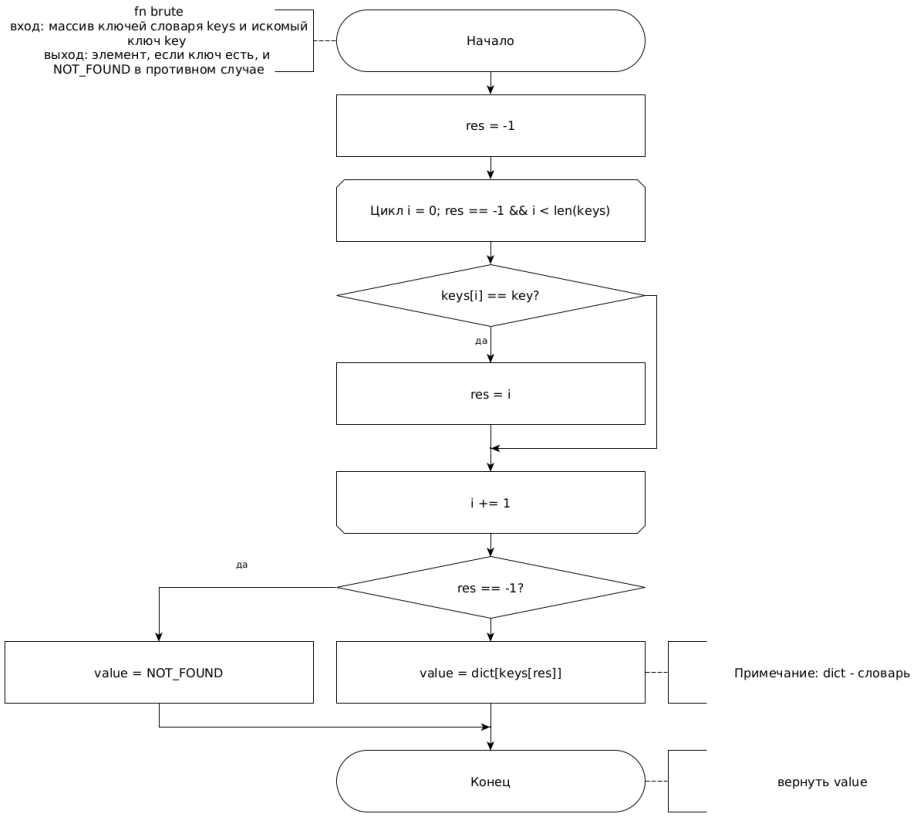

Рис. 2.1: Схема алгоритма поиска полным перебором.

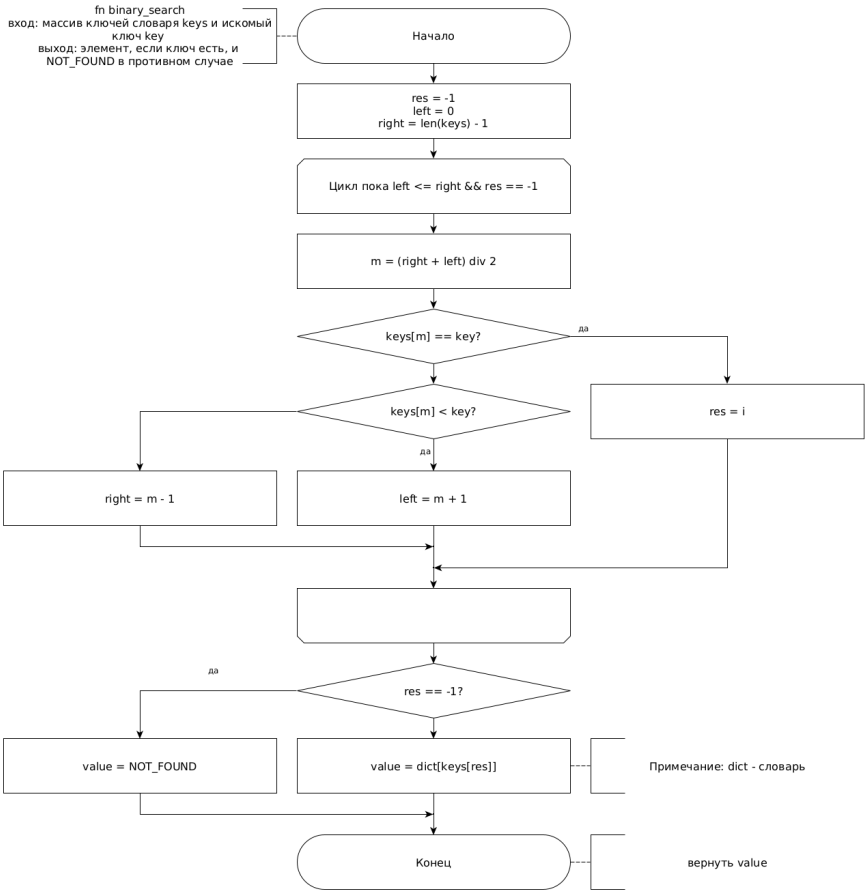

Рис. 2.2: Схема алгоритма поиска с использованием двоичного поиска.

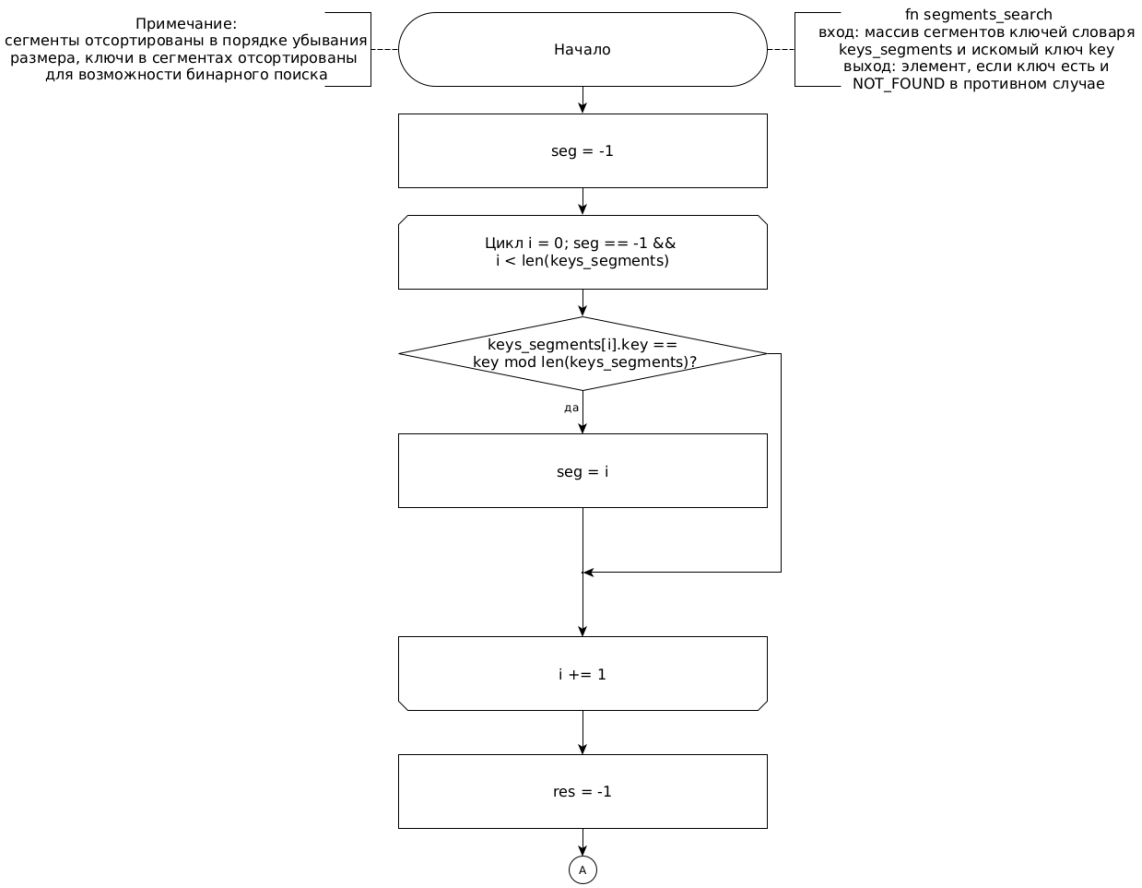

Рис. 2.3: Схема алгоритма поиска с использованием разделения на сегменты и частотного анализа.

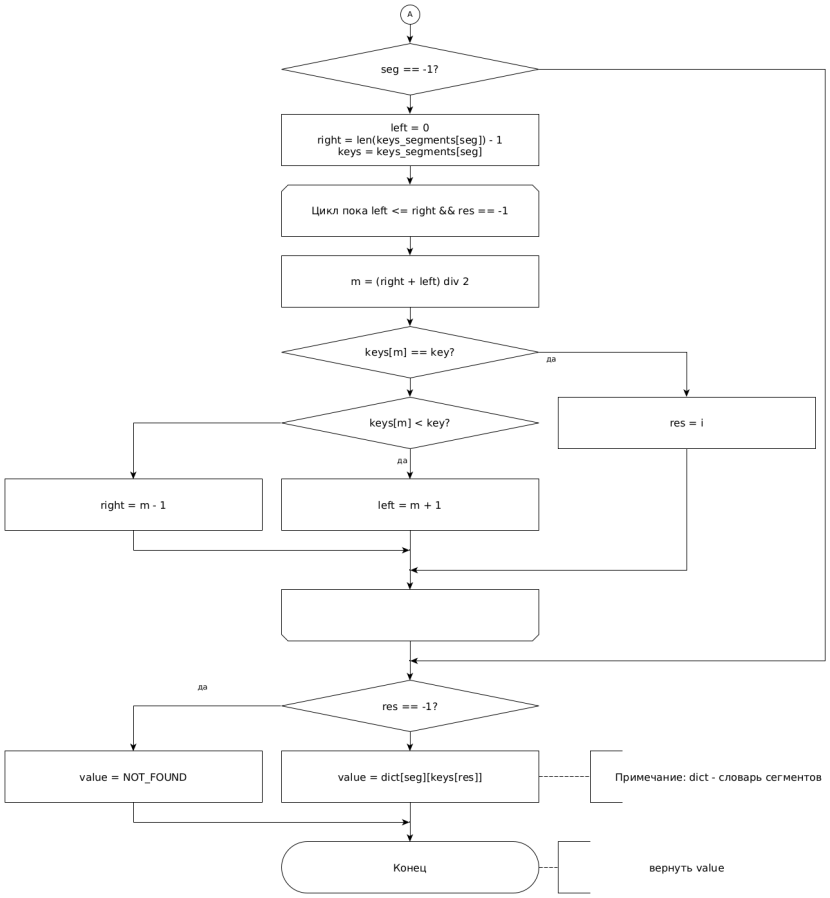

Рис. 2.4: Схема алгоритма поиска с использованием разделения на сегменты и частотного анализа. Продолжение.

Вывод

В данном разделе были рассмотрены структура словаря, на котором будут проводиться эксперименты, а также схемы алгоритмов поисков.

3 Технологическая часть

В данном разделе приведены средства программной реализации и листинг кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подается ключ;
- на выход программа выдает значение, хранящееся в словаре по ключу, если таковое присутствует, “пустое” значение в противном случае.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Rust [5]. Данный выбор обусловлен популярностью языка и скоростью его выполнения, а также тем, что данный язык предоставляет широкие возможности для написания тестов [6].

3.3 Листинг кода

В листинге 3.1 приведена реализация словарей.

```rust
use serde_derive::Deserialize;
use std::cmp::{Ord, Ordering};

pub type Id = usize;
pub type Name = String;

#[derive(Clone, Debug, Deserialize)]
pub struct DictEntry<K: Ord, V: Clone> {
    pub key: K,
    pub value: V,
}
```

```rust

pub trait Map<K: Ord, V: Clone> {
    fn get(&self, key: &K) -> Option<V>;
}

#[derive(Debug)]
pub struct BruteMap<K: Ord, V: Clone> {
    data: Vec<DictEntry<K, V>>,
}

impl<K: Ord, V: Clone> From<Vec<DictEntry<K, V>>> for BruteMap<K, V> {
    fn from(source: Vec<DictEntry<K, V>>) -> BruteMap<K, V> {
        BruteMap { data: source }
    }
}

impl<K: Ord, V: Clone> Map<K, V> for BruteMap<K, V> {
    fn get(&self, key: &K) -> Option<V> {
        self.data
            .iter()
            .find(|&e| e.key == *key)
            .map(|entry| entry.value.clone())
    }
}

pub struct BinaryMap<K: Ord, V: Clone> {
    data: Vec<DictEntry<K, V>>,
}

impl<K: Ord, V: Clone> From<Vec<DictEntry<K, V>>> for BinaryMap<K, V> {
    fn from(mut source: Vec<DictEntry<K, V>>) -> BinaryMap<K, V> {
        source.sort_by(|a, b| a.key.cmp(&b.key));
        BinaryMap { data: source }
    }
}

fn bsearch<T: Ord, U: Clone>(arr: &[DictEntry<T, U>], key: &T) -> Option<U> {
    let (mut left, mut right) = (0 as i64, arr.len() as i64 - 1);
    while left <= right {
        let m = ((left + right) / 2) as usize;
        match arr[m].key.cmp(&key) {
            Ordering::Less => left = m as i64 + 1,
            Ordering::Greater => right = m as i64 - 1,
            Ordering::Equal => return Some(arr[m].value.clone()),
        }
    }
    None
}
```

```
60
61 impl<K: Ord, V: Clone> Map<K, V> for BinaryMap<K, V> {
62     fn get(&self, key: &K) -> Option<V> {
63         bsearch(&self.data, key)
64     }
65 }
66
67 pub struct SegmentMap<G: Ord, K: Ord + Clone, V: Clone> {
68     data: Vec<DictEntry<G, Vec<DictEntry<K, V>>>>,
69     f: fn(&K) -> G,
70 }
71
72 pub trait SegMap<G: Ord, K: Ord, V> {
73     fn get(&self, key: &K) -> Option<V>;
74 }
75
76 impl<G: Ord, K: Ord + Clone, V: Clone> SegmentMap<G, K, V>
77 where
78     G: Ord,
79     K: Ord,
80 {
81     fn put(&mut self, pair: DictEntry<K, V>) {
82         let seg_key = (self.f)(&pair.key);
83         let inner = match self.data.iter().position(|e| e.key == seg_key) {
84             Some(p) => &mut self.data[p].value,
85             None => {
86                 self.data.push(DictEntry {
87                     key: seg_key,
88                     value: Vec::new(),
89                 });
90                 let index = self.data.len() - 1;
91                 &mut self.data[index].value
92             }
93         };
94         inner.push(pair);
95     }
96
97     pub fn from(source: Vec<DictEntry<K, V>>, f: fn(&K) -> G) -> SegmentMap<G, K, V> {
98         let mut m = SegmentMap {
99             f,
100             data: Vec::new(),
101         };
102         for e in source {
103             m.put(e);
104         }
105         m.data.sort_by(|a, b| b.value.len().cmp(&a.value.len()));
106         m
107     }
```

```
}

impl<G: Ord, K: Ord + Clone, V: Clone> SegMap<G, K, V> for SegmentMap<G, K, V> {
    fn get(&self, key: &K) -> Option<V> {
        let seg_key = (self.f)(&key);
        let inner = match self.data.iter().position(|e| e.key == seg_key) {
            Some(pos) => &self.data[pos].value,
            None => return None,
        };
        bsearch(inner, &key)
    }
}

impl<K: Ord + Clone, V: Clone> Map<K, V> for SegmentMap<K, K, V> {
    fn get(&self, key: &K) -> Option<V> {
        <SegmentMap<K, K, V> as SegMap<K, K, V>>::get(&self, key)
    }
}

#[cfg(test)]
mod tests;
```

Листинг 3.1: Реализация словарей.

3.4 Тестирование функций.

В таблице 3.1 представлены данные для тестирования. Все тесты пройдены успешно.

Ключ	Словарь	Ожидание	Результат
1	{1: "Иван М.", 2: "Олег К."}	"Иван М."	"Иван М."
3	{1: "Иван М.", 2: "Олег К."}	NOT_FOUND	NOT_FOUND
1	{}	NOT_FOUND	NOT_FOUND

Таблица 3.1: Тестирование функций.

Вывод

Была разработана и протестирована реализация словарей.

4 Исследовательская часть

В данном разделе приведены примеры работы программы и анализ характериситик разработанного программного обеспечения.

4.1 Технические характеристики

- Операционная система: Manjaro [7] Linux [8] x86_64.
- Память: 8 ГБ.
- Процессор: Intel® Core™ i7-8550U[9].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Замеры и исследование результатов.

Тестирование проводилось на 2 наборах для каждого алгоритма и каждого количества подборов. Первый набор включал в себя исключительно ключи, которые есть в словаре, причём данные ключи перебираются итеративно, что имитирует равновозможность выпадения ключа (в тестовом наборе 2409 ключей, таким образом для 10.000.000 подборов каждый ключ берется ≈ 4151 раз). Второй набор включал исключительно отсутствующие в словаре ключи, что позволяло проверить, насколько быстро алгоритмы поиска способны обнаружить тот факт, что ключа нет среди имеющихся.

В таблицах 4.1, 4.2 и 4.3 представлены времена работы полного перебора, двоичного поиска и сегментированного алгоритмов. В таблице представлены значения:

- количество раз, которое производился поиск (Кол-во);
- суммарное время поиска только присутствующих ключей (ВППК; в нс);

- суммарное время поиска только отсутствующих ключей (ВППК; в нс).

Примечание: для сегментированного алгоритма было выбрано деление на 5 сегментов.

Кол-во	ВППК	ВПОК
1000	16304689	14466108
10000	303890897	327973602
100000	3047600809	3473714374
1000000	33328461139	34389768377
10000000	341210360139	354735971458

Таблица 4.1: Время работы полного перебора.

Кол-во	ВППК	ВПОК
1000	410665	398989
10000	4132803	3969430
100000	40829389	40449005
1000000	407241884	409383404
10000000	4045076157	4065069524

Таблица 4.2: Время работы двоичного поиска.

Кол-во	ВППК	ВПОК
1000	491870	501553
10000	4205062	4243667
100000	42044052	43403032
1000000	406392543	408692222
10000000	4062301277	4084322643

Таблица 4.3: Время работы сегментированного алгоритма с частотным анализом.

На графике 4.1 построены графики зависимости времени от количества подборов из словаря для алгоритмов поиска (для данных из таблиц 4.1, 4.2 и 4.3):

- полного перебора при запросе только присутствующих ключей (BruteGood);

- полного перебора при запросе только отсутствующих ключей (BruteBad);
- бинарного при запросе только присутствующих ключей (BinaryGood);
- бинарного при запросе только отсутствующих ключей (BinaryBad);
- сегментированного с частотным анализом при запросе только присутствующих ключей (SegmentGood);
- сегментированного с частотным анализом при запросе только отсутствующих ключей (SegmentBad).

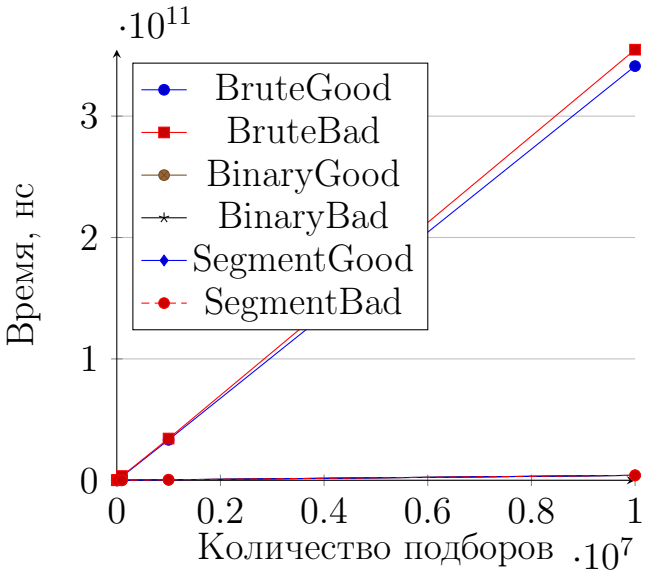

Рис. 4.1: Сравнение алгоритмов.

Вывод

Алгоритм полного перебора оказался самым медленным, при 10.000.000 элементах и поиске только присутствующих ключей среднее время поиска ≈ 34.121 мкс, в то время как среднее время поиска сегментированного алгоритма на тех же данных составило ≈ 0.406 мкс, что приблизительно равно времени бинарного поиска ≈ 0.405 мкс, который оказался менее чем на 1% быстрее. При том же наборе, но поиске только отсутствующих ключей полный перебор работает медленнее на ≈ 35.474 мкс (что составляет примерно 4%), в то время как времена сегментированного алгоритма

(≈ 0.408) и бинарного поиска (≈ 0.406), также ухудшились, но менее, чем на 1%.

Заключение

В рамках выполнения работы были выполнены следующие задачи:

- был реализован алгоритм поиска по словарю, использующий полный перебор;
- был реализован алгоритм поиска по словарю, использующий двоичный поиск;
- был применён частотный анализ для эффективного поиска по словарю;
- были сравнены результаты работы алгоритмов;
- были сделаны выводы по проделанной работе.

Работа показала, что алгоритм поиска полным перебором, работающий за линейное время, в общем случае на несколько порядков медленнее алгоритмов поиска, работающих за логарфмическое время.

Литература

[1] NIST. associative array. Режим доступа: https://xlinux.nist.gov/dads/HTML/assocarray.html (дата обращения: 08.12.2020).

[2] Cormen T. H. Introduction to Algorithms // MIT Press. 2001. p. 1292.

[3] Коршунов Ю. М. Коршуном Ю. М. Математические основы кибернетики // Энергоатомиздат. 1972.

[4] Official Site of the National Hockey League. Режим доступа: https://www.nhl.com/ (дата обращения: 02.10.2020).

[5] Rust Programming Language [Электронный ресурс]. URL: https://doc.rust-lang.org/std/index.html.

[6] Документация по ЯП Rust: бенчмарки [Электронный ресурс]. Режим доступа: https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html (дата обращения: 10.10.2020).

[7] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: https://manjaro.org/ (дата обращения: 10.10.2020).

[8] Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: https://www.linux.org.ru/ (дата обращения: 10.10.2020).

[9] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html (дата обращения: 10.10.2020).