Kauno technologijos universitetas

Informatikos fakultetas

Laboratorinis darbas Nr. 3

Ataskaita

Aistis Jakutonis

Studentas

Doc. Butkevičiūtė Eglė

Dėstytoja

Kaunas, 2024

Turinys

Greitaveika

Tiriamieji metodai:

12	Class HashMapOa: contains()	Class java.util.HashMap <E>: contains()

Aprašytas contains() metodas HashMapOa klasėje:

```
@Override
    public boolean contains(K key) {
        if (key == null) {
            throw new IllegalArgumentException("Key is null in contains(K
key)");
        }

        return get(key) != null;
    }
```

Aprašytas containsKey() metodas java.util.HashMap klasėje

```
public boolean containsKey(Object key) {
        return getNode(hash(key), key) != null;
    }
```

Metodo sudėtingumas naudojant HashMapOa klasę:

Šio metodo sudėtingumas Yra O(n). Taip yra todėl, nes šis metodas blogiausiu atveju turi iš eilės pereiti per visus elementus prieš rasdamas reikiamą.

Metodo sudėtingumas naudojant java.util.HashMap klasę:

Šio metodo taip pat yra log2(n) blogiausiu atveju dėl tos, nes šis metodas naudoja raudonai juodą medį. Ši klasė buvo tobulinama kelis metus ir yra parašyta sklandžiau įvertinant alternatyvas.

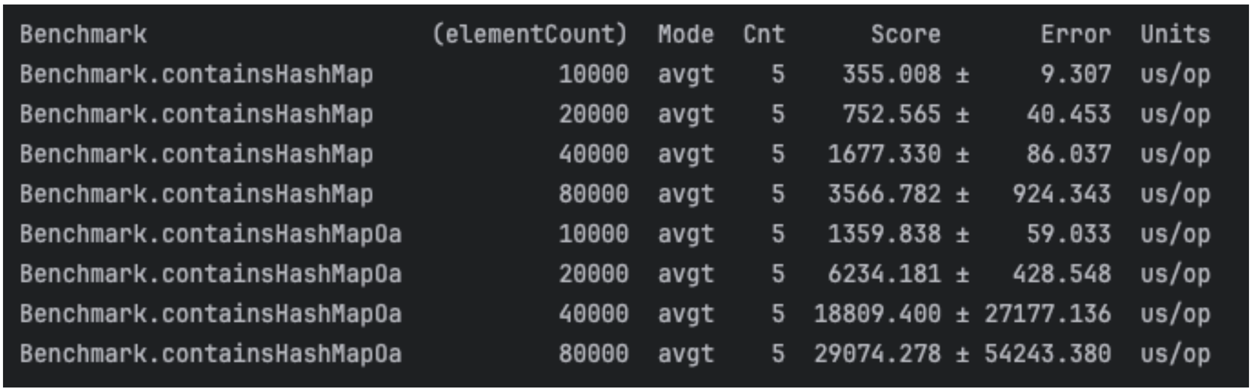
Greitaveikos testavimo kodas:

```
package edu.ktu.ds.lab3.demo;

import edu.ktu.ds.lab3.utils.HashManager;
import edu.ktu.ds.lab3.utils.HashMap;
import edu.ktu.ds.lab3.utils.HashMapOa;
import edu.ktu.ds.lab3.utils.Map;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.BenchmarkParams;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
```

```
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@State(Scope.Benchmark)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(time = 1, timeUnit = TimeUnit.SECONDS)
public class Benchmark {

    @State(Scope.Benchmark)
    public static class FullMap {

        List<String> ids;
        List<Car> cars;
        HashMap<String, Car> carsMap;

        @Setup(Level.Iteration)
        public void generateIdsAndCars(BenchmarkParams params) {
            ids =
Benchmark.generateIds(Integer.parseInt(params.getParam("elementCount")));
            cars =
Benchmark.generateCars(Integer.parseInt(params.getParam("elementCount")));
        }

        @Setup(Level.Invocation)
        public void fillCarMap(BenchmarkParams params) {
            carsMap = new HashMap<>(HashManager.HashType.DIVISION);
            putMappings(ids, cars, carsMap);
        }
    }

    @Param({"10000", "20000", "40000", "80000"})
    public int elementCount;

    List<String> ids;
    List<Car> cars;

    @Setup(Level.Iteration)
    public void generateIdsAndCars() {
        ids = generateIds(elementCount);
        cars = generateCars(elementCount);
    }

    static List<String> generateIds(int count) {
        return new ArrayList<>(CarsGenerator.generateShuffleIds(count));
    }

    static List<Car> generateCars(int count) {
        return new ArrayList<>(CarsGenerator.generateShuffleCars(count));
    }

    @org.openjdk.jmh.annotations.Benchmark
    public boolean containsHashMapOa(){
        Map<String, Car> map = new HashMapOa<>(HashManager.HashType.DIVISION);
        putMappings(ids, cars, map);
```

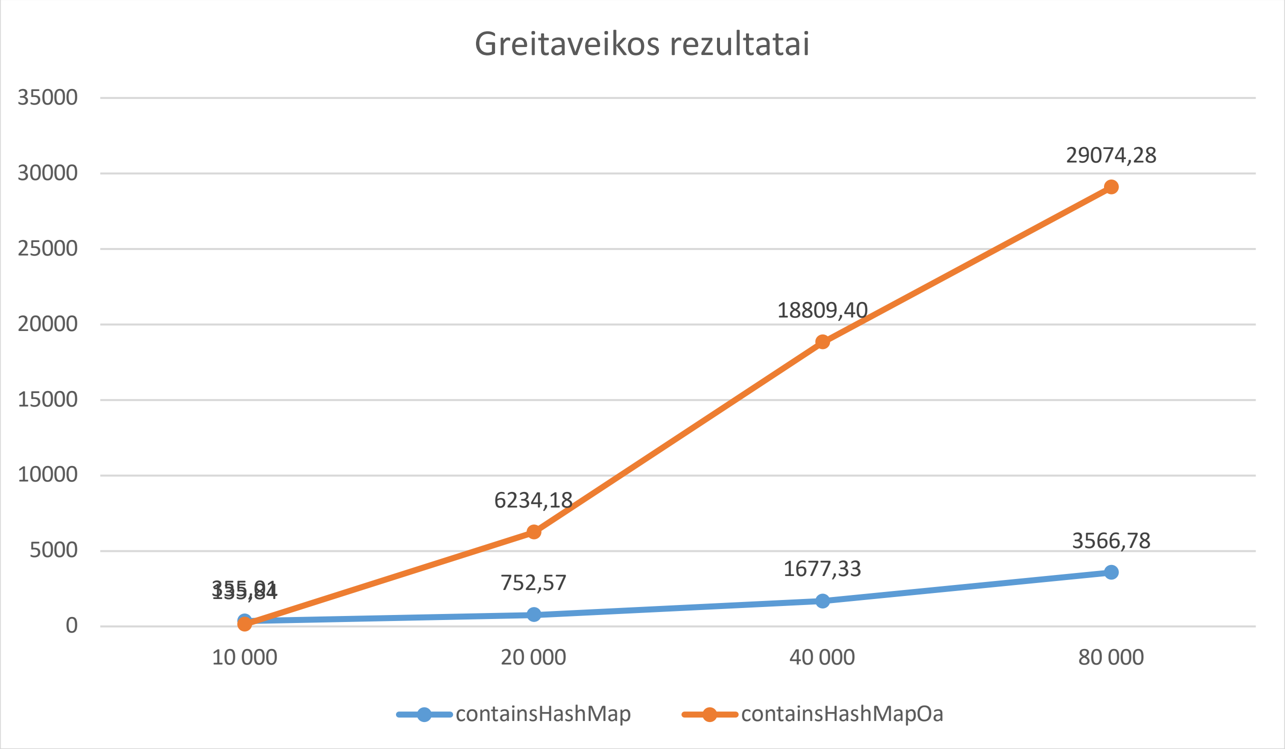
```
        for (String id : ids) {
            if (map.contains(id)) {
                return true;
            }
        }
        return false;
    }

    @org.openjdk.jmh.annotations.Benchmark
    public boolean containsHashMap() {
        java.util.Map<String, Car> map = new java.util.HashMap<>();
        putMappingsJava(ids, cars, map);

        for (String id : ids) {
            if (map.containsKey(id)) {
                return true;
            }
        }
        return false;
    }

    public static void putMappingsJava(List<String> ids, List<Car> cars,
java.util.Map<String, Car> carsMap) {
        for (int i = 0; i < cars.size(); i++) {
            carsMap.put(ids.get(i), cars.get(i));
        }
    }

    public static void putMappings(List<String> ids, List<Car> cars, Map<String,
Car> carsMap) {
        for (int i = 0; i < cars.size(); i++) {
            carsMap.put(ids.get(i), cars.get(i));
        }
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(Benchmark.class.getSimpleName())
                .forks(1)
                .build();
        new Runner(opt).run();
    }
}
```

Šis greitaveikos kodas yra gana nesudėtingas. Mašinų mapai yra sugeneruojami atskirai naudojant tam sukurtą generavimo klasę. Yra įvykdomi aprašyti metodai ir galiausiai atspausdinami gauti greitaveikos rezultatai:



```
Benchmark                     (elementCount)  Mode  Cnt      Score       Error  Units
Benchmark.containsHashMap              10000  avgt    5    355.008 ±     9.307  us/op
Benchmark.containsHashMap              20000  avgt    5    752.565 ±    40.453  us/op
Benchmark.containsHashMap              40000  avgt    5   1677.330 ±    86.037  us/op
Benchmark.containsHashMap              80000  avgt    5   3566.782 ±   924.343  us/op
Benchmark.containsHashMapOa            10000  avgt    5   1359.838 ±    59.033  us/op
Benchmark.containsHashMapOa            20000  avgt    5   6234.181 ±   428.548  us/op
Benchmark.containsHashMapOa            40000  avgt    5  18809.400 ± 27177.136  us/op
Benchmark.containsHashMapOa            80000  avgt    5  29074.278 ± 54243.380  us/op
```

Grafikas:



Kompiuterio parametrai, kuriame buvo atlikti greitaveikos testavimai:
Procesoriaus charakteristikos – Apple M1 Max
Atminties kiekis (RAM) – 32GB
OS – macOS Sonoma 14.4.1

Rezultatuose svarbu atsižvelgti į elementų kiekio stulpelį bei į „score“ stulpelį, kuriame surašyti greičiai mikrosekundėmis. Kaip matome, kad metodų įvykdymo greičiai didėja kartu didėjant ir elementų skaičiui. Iš šių rezultatų akivaizdžiai matosi, kad naudojant java.util.HashMap klasės containsKey metodą viskas yra įvykdomą žymiai greičiau nei tai atliekant su HashMapOa klase. Taip yra todėl, nes ši javos klasė yra ištobulinta ir sugeba įvertinti daugiau alternatyvų, kas sumažina papildomas laiko sąnaudas. Beto ši klasė gali būti, kad naudoja kitokią hash funkciją, kas taip pat gali įtakoti jos veikimą.