Duomenų struktūrų lab3 ataskaita

Parengė:

Vytenis Kriščiūnas IFF-1/1

Tiriamieji metodai

Class HashMapOa: containsValue() ir Class java.util.HashMap <E>: containsValue().

Klasės HashMapOa: containsValue() metodas:

```
3 usages
public boolean containsValue(Object value) {
    for (int i = 1; i < table.length; i++){
        if (table[i] != null && table[i].value != null && table[i].value.equals(value)){
            return true;
        }
    }
    return false;
}
```

Metodas patikrina ar atvaizdyje egzistuoja vienas ar daugiau raktų metodo argumente nurodytai reikšmei. Klasės java.util.HashMap <E>: containsValue() metodas atlieka tą pačia paieška panašiu principu.

Asimptotinis sudėtingumas

Class HashMapOa: containsValue()

Asimpttotinis sudėtingumas O(n).

Class java.util.HashMap <E>: containsValue()

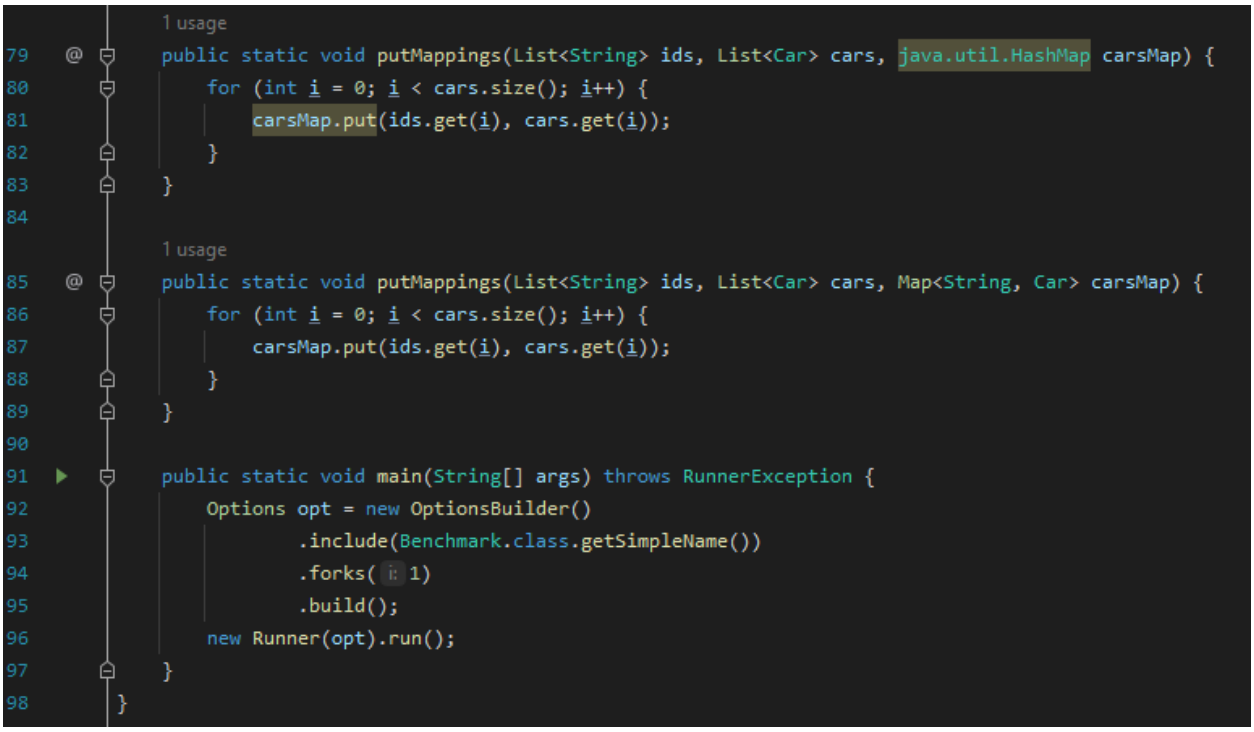
Asimpttotinis sudėtingumas $O(n^2)$.

Greitaveikos testavimas

Testavimo klasė:

```java
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;

16 usages  4 inheritors
@BenchmarkMode(Mode.AverageTime)
@State(Scope.Benchmark)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(time = 1, timeUnit = TimeUnit.SECONDS)
public class Benchmark {

    7 usages  4 inheritors
    @State(Scope.Benchmark)
    public static class FullMap {

        3 usages
        List<String> ids;
        5 usages
        List<Car> cars;
        3 usages
        java.util.HashMap carsMap;

        3 usages
        HashMapOa<String, Car> carsMapOa;

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

```java
    20 usages
    @Setup(Level.Invocation)
    public void fillCarMap(BenchmarkParams params) {
        carsMap = new java.util.HashMap<>(HashManager.HashType.DIVISION.ordinal());
        carsMapOa = new HashMapOa(HashManager.HashType.DIVISION);
        putMappings(ids, cars, carsMap);
        putMappings(ids, cars, carsMapOa);
    }
}

4 usages
@Param({"4000", "8000", "16000", "32000", "64000"})
public int elementCount;

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

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

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

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

10 usages
@org.openjdk.jmh.annotations.Benchmark
public void MapContainsValue(FullMap fullMap) {
    fullMap.cars.forEach(car -> fullMap.carsMap.containsValue(car));
}

10 usages
@org.openjdk.jmh.annotations.Benchmark
public void MapOaContainsValue(FullMap fullMap) {
    fullMap.cars.forEach(car -> fullMap.carsMapOa.containsValue(car));
}
```



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

1 usage
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( i: 1)
            .build();
    new Runner(opt).run();
}
}
```

Pasirenku elementų kiekius testavimui. Sukuriu elementus pagal pasirinktą skaičių ir juos sudedu į java.util.HashMap ir HashMapOa kintamuosius. Tada apsirašiau du metodus containsValue(), kuriuose naudoju containsValue() metodus. Galiausiai pradėjus greitaveikos testavimą stebiu gautus rezultatus.

Kompiuterio parametrai

Procesorius:

AMD FX-6300 six-core, greitis – 3.50 Ghz., 3 branduoliai, apdorojimas – 64-bit, talpykla – 8 MB.

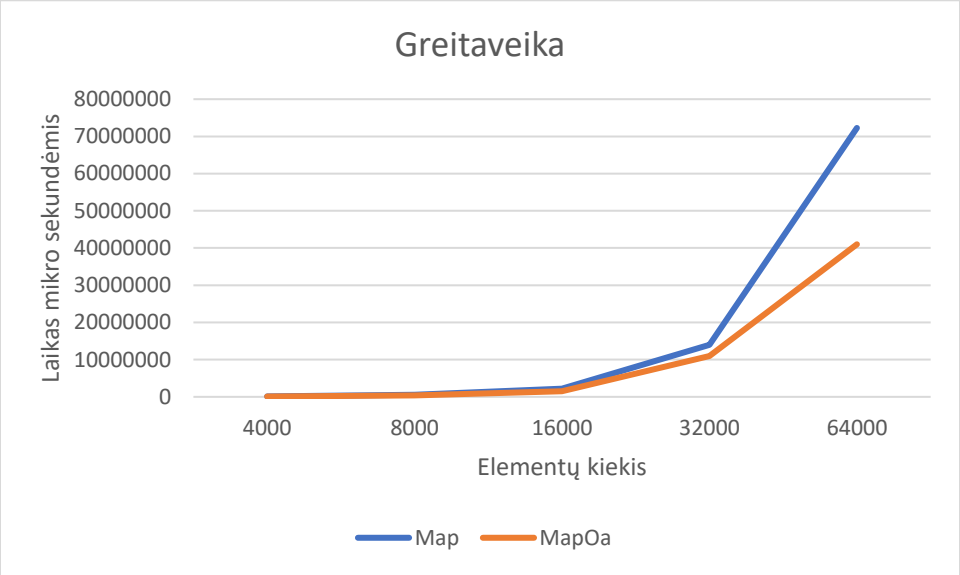
Atmintis:

16 GB

Talpa:

SSD 224 GB

Algoritmų/ metodų vykdymo laiko priklausomybės nuo įvesties duomenų kiekio grafikas



Išvados

Pagal gautus rezultatus akivaizdu, kad HashMapOa klasės containsValue() metodas yra greitesnis nei java.util.HashMap klasės containsValue() metodas. Nors šių klasių metodų asimptotiniai sudėtingumai pagal įgyvendinimą skiriasi, tačiau teorinėje medžiagoje rašo, kad jų simptotiniai sudėtingumai nesiskiria. Kuo didesnis elementų kiekis tuo didesnis greičių skirtumas yra matomas.