****

**Kauno technologijos universitetas**

Informatikos fakultetas

**Laboratorinis darbas Nr. 2**

Ataskaita

|  |  |
| --- | --- |
|  |  |
| **Aistis Jakutonis**  Studentas | (parašas) (data) |
|  |  |
| **Doc. Butkevičiūtė Eglė**  Dėstytoja | (parašas) (data) |
|  |  |

**Kaunas, 2024**

Turinys

[Greitaveika 3](#_Toc181221937)

Greitaveika

Tiriamieji metodai:



Aprašytas addAll metodas BstSet klasėje:

/\*\*

\* Abės set elementai pridedami į esamą aibę, jeigu abi aibės turi tą patį elementą, jis nėra dedamas.

\*

\* @param set pridedamoji aibė

\*/

**@Override**

**public** **void** **addAll**(Set<E> set) {

**for** (E element : set) {

add(element);

}

}

Šio metodo nereikėjo aprašinėti AvlSet klasėje, nes ji paveldi BstSet klasę.

Metodo sudėtingumas naudojant BstSet klasę:

Tarkime, kad m yra elementų skaičius abėje Set. Tai šio metodo sudėtingumas bus: m \* n. Taip yra todėl, nes mūsų addAll metodas pirmiausia eina per kiekvieną m elementą atskirai ir taip pat kiekvienam m elementui jis atlieka add metodą, kuris naudoja rekursinį naujo elemento pridėjimą į medį, todėl add metodo sudėtingumas yra log(n). Šių metodų sudėtingumus sudaugine gauname rezultatą.

Metodo sudėtingumas naudojant AvlSet klasę:

Šio metodo sudėtingumas bus m \* log(n), nes jis yra subalancuotas. Tačiau kartais atrodytų, kad balansavimo dalis addRecursive metode turėtų padidinti sudėtingumą. Vis dėlto taip nėra, nes balansavimo operacija neužtrunka taip ilgai, nes ji vyksta tik tarp kelių susijusių mazgų.

Greitaveikos testavimo kodas:

**package** edu.ktu.ds.lab2.demo;

**import** **edu.ktu.ds.lab2.utils.AvlSet**;

**import** **edu.ktu.ds.lab2.utils.BstSet**;

**import** **edu.ktu.ds.lab2.utils.BstSetIterative**;

**import** **edu.ktu.ds.lab2.utils.SortedSet**;

**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.HashSet**;

**import** **java.util.Set**;

**import** **java.util.concurrent.TimeUnit**;

**@BenchmarkMode**(Mode.AverageTime)

**@State**(Scope.Benchmark)

**@OutputTimeUnit**(TimeUnit.MICROSECONDS)

**@Warmup**(time = **1**, timeUnit = TimeUnit.SECONDS, iterations = **2**)

**@Measurement**(time = **1**, timeUnit = TimeUnit.SECONDS)

**public** **class** **Benchmark** {

**@State**(Scope.Benchmark)

**public** **static** **class** **FullSet** {

Car[] cars;

BstSet<Car> carSet;

AvlSet<Car> avlSet;

**@Setup**(Level.Iteration)

**public** **void** **generateElements**(BenchmarkParams params) {

cars = Benchmark.generateElements(Integer.parseInt(params.getParam("elementCount")));

}

**@Setup**(Level.Invocation)

**public** **void** **fillCarSet**(BenchmarkParams params) {

carSet = **new** BstSet<>(Car.byPrice);

avlSet = **new** AvlSet<>(Car.byPrice);

addElements(cars, carSet);

addElements(cars, avlSet);

}

}

**@Param**({"1000", "2000", "5000", "10000"})

**public** **int** elementCount;

Car[] cars;

BstSet<Car> carSet2;

**@Setup**(Level.Iteration)

**public** **void** **generateElements**() {

cars = generateElements(elementCount);

carSet2 = **new** BstSet<>(Car.byPrice);

addElements(cars, carSet2);

}

**static** Car[] **generateElements**(**int** count) {

**return** **new** **CarsGenerator**().generateShuffle(count, **1.0**);

}

**@org.openjdk.jmh.annotations.Benchmark**

**public** **void** **addAllBstSet**(FullSet fullSet) {

fullSet.carSet.addAll(carSet2);

}

**@org.openjdk.jmh.annotations.Benchmark**

**public** **void** **addAllAvlSet**(FullSet fullSet) {

fullSet.avlSet.addAll(carSet2);

}

**public** **static** **void** **addElements**(Car[] carArray, SortedSet<Car> carSet) {

**for** (Car car : carArray) {

carSet.add(car);

}

}

**public** **static** **void** **main**(String[] args) **throws** RunnerException {

Options opt = **new** OptionsBuilder()

.include(Benchmark.class.getSimpleName())

.forks(**1**)

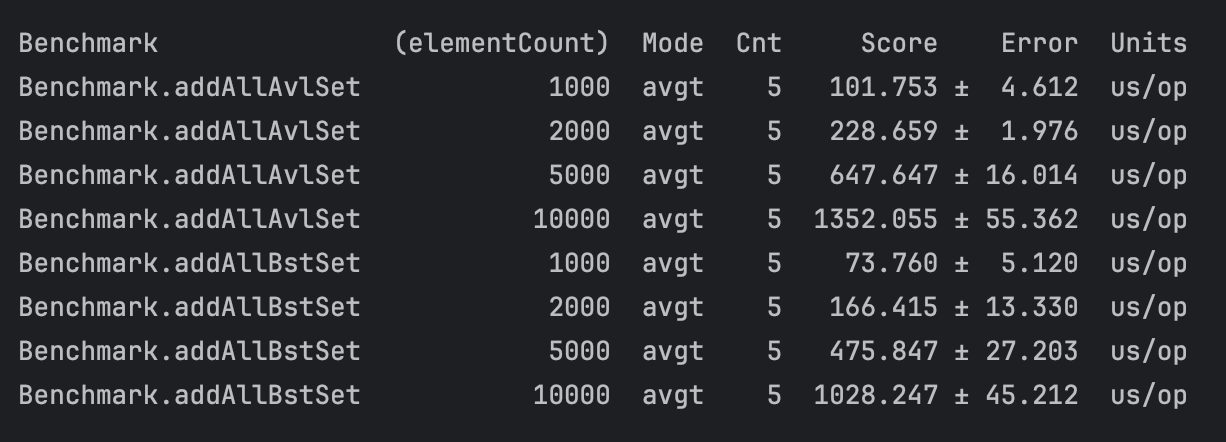
.build();

**new** **Runner**(opt).run();

}

}

Šis greitaveikos kodas yra gana nesudėtingas. Mašinų dvejetainiai medžiai yra atskirai sugeneruojami naudojami tam sukurtą generavimo klasę. Taip pat atskirai yra sukuriamas addAll metodo parametro medis. Yra įvykdomi aprašyti metodai ir galiausiai atspausdinami gauti greitaveikos rezultatai:



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 ne logoritmiškai. Taip yra todėl, kad pasirinktas elementų kiekis buvo kiek per mažas. Tačiau norint įrodyti logoritmišką sudėtingumą reikėtų turėti itin daug elementų. Tačiau iš šių rezultatų akivaizdžiai matosi, kad naudojant bst medį addAll metodas yra įvykdomas greičiau nei naudojant avl medį. Taip yra todėl, nes įdedant kiekvieną elementą į avl medį, jį reikia dar ir balansuoti, tikrinti ar jis neišsibalansavo. Šis veiksmas prailgina šio metodo darbą ir tai atsiliepia jo atlikimo greityje.