Duomenų struktūrų technologinio projekto – *Splay tree* ataskaita

Parengė:

Vytenis Kriščiūnas IFF-1/1

Realizuojama duomenų struktūra

Naudojami yra sukurto automobilio duomenys:

```
private static final int minYear = 2000;
2 usages
private static final int currentYear = LocalDate.now().getYear();
2 usages
private static final double minPrice = 100.0;
2 usages
private static final double maxPrice = 333000.0;


2 usages
private static final String idCode = "TA";   //  ***** nauja
2 usages
private static int serNr = 100;               //  ***** nauja

5 usages
private final String carRegNr;

3 usages
private String make = "";
3 usages
private String model = "";
5 usages
private int year = -1;
3 usages
private int mileage = -1;
5 usages
private double price = -1.0;
```

Šie duomenys yra saugomi splayTree<E> medyje mazgų (nodes) pagalba.

Siekiant vizualiai perteikti medį buvau naudojamas masyvas. Iteratoriaus realizavimui teko naudoti tiesinę duomenų struktūrą – Stack.

Esminė šios duomenų struktūros savybė yra splay operacija, kuri yra medžio „tvarkymo“ priemonė. Atliekant pagrindinius metodus (pridėjimo, paieškos, šalinimo), splay operacija perkele norimą elementą iki medžio šaknies, darydama tam tikras šešių tipų rotacijas.

Ši duomenų struktūra palaiko pridėjimo, paieškos, šalinimo, splay, dydžio paieškos, aukščio paieškos operacijas. Blogiausiu atveju šių operacijų asimptotiniai sudėtingumai bus O(n), o vidutiniu (tipišku) $O(\log_2 N)$.

Duomenų struktūros testavimas

Duomenų struktūra buvo testuojama ManualTest klasėje. Buvo kuriamos mašinos ir atsitiktine tvarka talpinamos splay tree struktūroje. Tada tikrinta ar elemento pridėjimo operacija teisingai formuoja medį.

Tikrinta šalinimo operacija panaikinus viena iš patalpintų automobilių. Taip pat svarbu įsitikinti, kad šalinimo operacija teisingai atliktų rotacijas ir būtų gaunas teisingai suformuotas medis.

Bandyta nustatyti, kas nutiktų, jei būtų talpinama jau egzistuojanti mašina. Ji nėra pridedama į medį, tačiau medžio struktūra keičiasi atliekant splay operaciją.

Ištestuotas paieškos metodas, bandant paimti iš medžio vieną iš automobilių. Tai atlikus ieškomas automobilis atsiduria šaknyje ir medis yra permaišomas.

Testuotas contains metodas, kuris tikrina ar mašina egzistuoja medyje. Atlikus šią operaciją medis yra permaišomas ir ieškota mašina atsidure šaknyje.

Rastas medžio dydis ir aukštis, kai kreiptasi į atitinkamus metodus.

Testavimų klasė ManualTest:

```
SplaySet<Car> set = new SplaySet<Car>();
Car c1 = new Car( make: "Renault", model: "Laguna", year: 2007, mileage: 50000, price: 1700);
Car c2 = new Car( make: "BMW", model: "E39", year: 2000, mileage: 10000, price: 3000);
Car c3 = new Car( make: "Tojota", model: "Elm", year: 2010, mileage: 200000, price: 10000);
Car c4 = new Car( make: "Subaru", model: "Auto", year: 2003, mileage: 50000, price: 2000);
Car c5 = new Car( make: "Volksvagen", model: "Pasat", year: 2007, mileage: 150000, price: 1500);
Car c6 = new Car( make: "Lexus", model: "Rib", year: 2005, mileage: 12000, price: 17000);

Ks.oun( obj: "Patalpinti duomenys");
set.put(c5);
set.put(c6);
set.put(c3);
set.put(c2);
set.put(c1);
set.put(c4);

for (Car c : set){
    Ks.oun(c);

}
Ks.oun( obj: "");
Ks.oun(set.toVisualizedString( dataCodeDelimiter: ""));

Ks.oun( obj: "Panaikintas automobilis Tojota");
set.remove(c3);
for (Car c : set){
    Ks.oun(c);

}
Ks.oun( obj: "");
Ks.oun(set.toVisualizedString( dataCodeDelimiter: ""));
```

```
Ks.oun( obj: "Bandoma įdėti jau egzistuojantį automobilį Renault");
set.put(c1);
for (Car c : set){
    Ks.oun(c);

}
Ks.oun( obj: "");
Ks.oun(set.toVisualizedString( dataCodeDelimiter: ""));

Ks.oun( obj: "Bandoma paimti automobilį BMW");
Ks.oun(set.get(c2));
Ks.oun( obj: "");
Ks.oun(set.toVisualizedString( dataCodeDelimiter: ""));

Ks.oun( obj: "Tikrinama ar egzistuoja automobilis Tojota");
Ks.oun(set.contains(c3));
Ks.oun( obj: "");
Ks.oun(set.toVisualizedString( dataCodeDelimiter: ""));

Ks.oun( obj: "Gražinamas splayTree elementų skaičius");
Ks.oun(set.size());
Ks.oun( obj: "");

Ks.oun( obj: "Gražinamas medžio šaknies aukštis");
Ks.oun(set.height());
```

Greitaveikos tyrimas

Greitaveikos testavimas vyko Benchmark klasėje. Buvo testuojami trijų medžių formavimo duomenų struktūrų: bst tree, avl tree ir splay tree paieškos, šalinimo ir pridėjimo metodai. Testavau skirtingus elementų kiekius, siekdamas išsiaiškinti, kaip pakis atitinkamų metodų testavimų laikai.

Pridėjimo algoritmai yra skirtingų duomenų struktūrų tipų, tikrinantys testavimo laikus skirtingoms duomenų struktūroms.

Šalinimo algoritmai yra void tipo, kurie šalina elementus vieną po kito iš atitinkamų duomenų struktūrų medžių.

Paieškos algorimtai yra void tipo, jie ieško atitinkamose duomenų stuktūrose patalpintų elementų.

Pateikta Benchmark klasė:

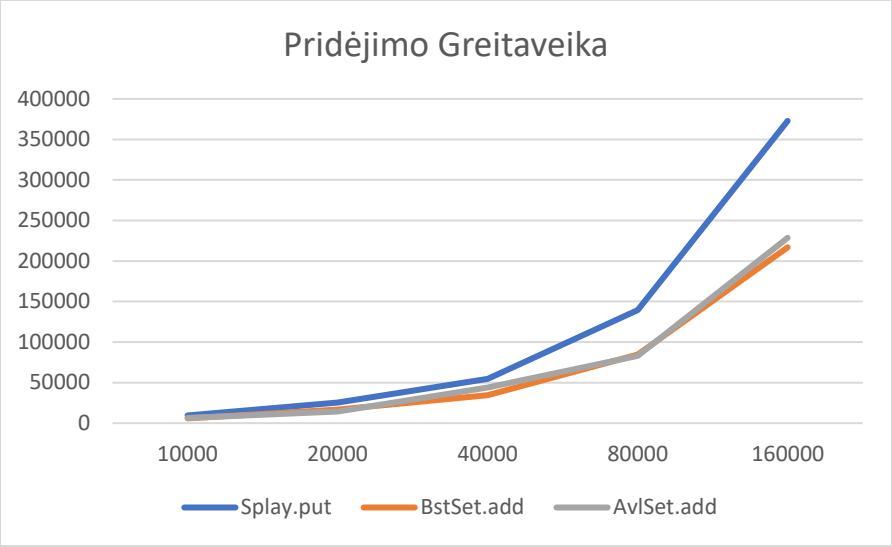
```java
package demo;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.BenchmarkParams;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.concurrent.TimeUnit;

8 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 {

    10 usages  8 inheritors
    @State(Scope.Benchmark)
    public static class FullSet {

        8 usages
        Car[] cars;
        4 usages
        SplaySet<Car> splayTree;

        5 usages
        BstSet<Car> bstSet;

        5 usages
        AvlSet<Car> avlSet;

        @Setup(Level.Iteration)
        public void generateElements(BenchmarkParams params) {
            cars = Benchmark.generateElements(Integer.parseInt(params.getParam( key: "elementCount")));
        }
```

```
    70 usages
    @Setup(Level.Invocation)
    public void fillCarSet(BenchmarkParams params) {
        splayTree = new SplaySet<Car>();
        addElements(cars, splayTree);

        bstSet = new BstSet<>();
        addElements(cars, bstSet);

        avlSet = new AvlSet<>();
        addElements(cars, avlSet);
    }
}

@Param({"1000", "2000", "4000", "8000", "16000"})
public int elementCount;

4 usages
Car[] cars;

@Setup(Level.Iteration)
public void generateElements() {
    cars = generateElements(elementCount);
}
2 usages
static Car[] generateElements(int count) {
    return new CarsGenerator().generateShuffle(count, shuffleCoef: 1.0);
}

///SplayTree tikrinimas
10 usages
@org.openjdk.jmh.annotations.Benchmark
public SplaySet<Car> SplayTreePut() {
    SplaySet<Car> splayTree = new SplaySet<Car>();
    addElements(cars, splayTree);
    return splayTree;
}

10 usages
@org.openjdk.jmh.annotations.Benchmark
public void SplayTreeRemove(FullSet carSet) {
    for (Car c : carSet.cars){
        carSet.splayTree.remove(c);
    }
}
```

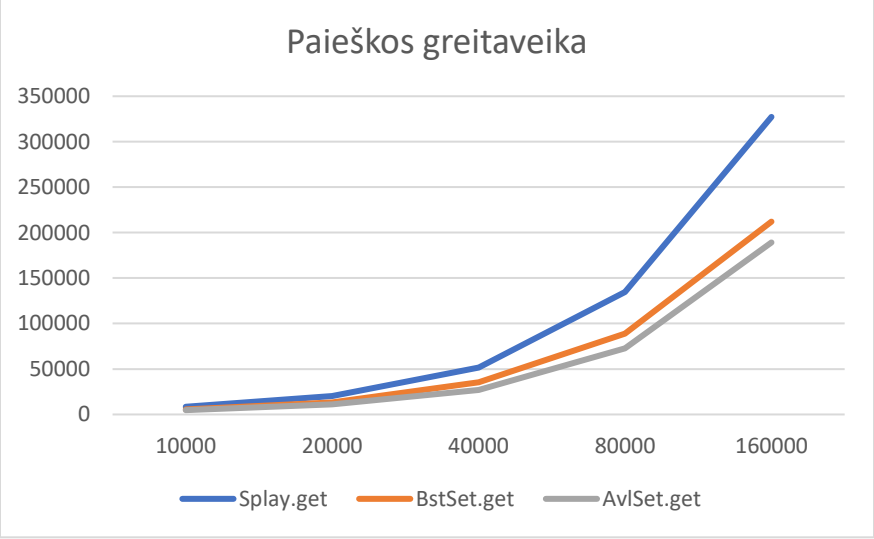
```
10 usages
@org.openjdk.jmh.annotations.Benchmark
public void SplayTreeGet(FullSet carSet) {
    for (Car c : carSet.cars){
        carSet.splayTree.get(c);
    }
}

///BstTree tikrinimas
10 usages
@org.openjdk.jmh.annotations.Benchmark
public BstSet<Car> BstSetAdd(){
    BstSet<Car> bstSet = new BstSet<>();
    addElements(cars, bstSet);
    return bstSet;
}

@org.openjdk.jmh.annotations.Benchmark
public void BstSetRemove(FullSet carSet){
   for (Car c : carSet.bstSet){
       carSet.bstSet.remove(c);
   }
}

@org.openjdk.jmh.annotations.Benchmark
public void BstSetGet(FullSet carSet) {
    for (Car c : carSet.cars){
        carSet.bstSet.get(c);
    }
}

///AvlTree tikrinimas
10 usages
@org.openjdk.jmh.annotations.Benchmark
public BstSet<Car> AvlSetAdd(){
    AvlSet<Car> avlSet = new AvlSet<>();
    addElements(cars, avlSet);
    return avlSet;
}

@org.openjdk.jmh.annotations.Benchmark
public void AvlSetRemove(FullSet carSet){
    for (Car c : carSet.avlSet){
        carSet.avlSet.remove(c);
    }
}
```

```java
    @org.openjdk.jmh.annotations.Benchmark
    public void AvlSetGet(FullSet carSet) {
        for (Car c : carSet.cars){
            carSet.avlSet.get(c);
        }
    }

    2 usages
    public static void addElements(Car[] carArray, SplaySet<Car> carSet) {
        for (int i = 0; i < carArray.length; i++) {
            carSet.put(carArray[i]);
        }
    }

    2 usages
    public static void addElements(Car[] carArray, BstSet<Car> carSet) {
        for (int i = 0; i < carArray.length; i++) {
            carSet.add(carArray[i]);
        }
    }

    2 usages
    public static void addElements(Car[] carArray, AvlSet<Car> carSet) {
        for (int i = 0; i < carArray.length; i++) {
            carSet.add(carArray[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();
    }
}
```

Laiko priklausomybės nuo įvesties duomenų kiekio grafai


Pridėjimo Greitaveika
400000
350000
300000
250000
200000
150000
100000
50000
0
10000
20000
40000
80000
160000
Splay.put
BstSet.add
AvlSet.add


Šalinimo greitaveika
400000
350000
300000
250000
200000
150000
100000
50000
0
10000
20000
40000
80000
160000
Splay.remove
BstSet.remove
AvlSet.remove


Paieškos greitaveika
350000
300000
250000
200000
150000
100000
50000
0
10000
20000
40000
80000
160000
Splay.get
BstSet.get
AvlSet.get

Išvados

Užduotyje įgyvendinta splay tree duomenų struktūra veikia pagal reikalavimus – įgyvendina reikalingus metodus teisingai. Ištestuotų splay tree paieškos, šalinimo, pridėjimo metodų eksperimentiniai asimptotiniai sudėtingumai visada buvo O(N), tai atitinka blogiausią sudėtingumų teorinį atvėjį. Bst duomenų struktūros eksperimentinis asimptotinis sudėtingumas nesiskyrė nuo teorinio ir buvo O(N), o avl duomenų struktūros skyrėsi. Nors avl turėtų visada realizuoti $O(\log_2 N)$ asimptotinį sudėtingumą, eksperimentiškai nustatyta, kad sudėtingumas yra O(N). Visi ištestuoti avl metodai laiko požiūriu buvo geresni palyginus su kitomis duomenų struktūromis, taip yra todėl, kad avl medis yra balansuotas ir jo balansavimas užtrunka trumpiau negu splay tree medžio, kuris yra balansuojamas visada. Avl medžio teorinis asimptotinis sudėtingumas yra geriausias, nes jis visada yra $O(\log_2 N)$, todėl ši duomenų struktūra veikia sparčiausiai.