

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**DUOMENŲ STRUKTŪROS**

**3 LABORATORINIS DARBAS**

**Atliko Karolis Stasaitis IFF8-9**

**Pagrindinė Klasė:**

package edu.ktu.ds.lab3.cpu;  
  
import edu.ktu.ds.lab3.utils.Ks;  
import edu.ktu.ds.lab3.utils.Parsable;  
  
import java.util.\*;  
  
public class CPU implements Parsable<CPU> {  
  
 private String CPU\_NAME;  
 private int CPU\_RELEASE\_YEAR;  
 private int CPU\_CORES;  
 private double CPU\_BASE\_MIN;  
 private double CPU\_BASE\_MAX;  
 private double CPU\_PRICE;  
  
 private static String *CPU\_REG\_PREF* = "ID";  
 private static int *CPU\_NUMBER* = 100;  
 private static String *CPU\_ID*;  
  
  
 public CPU()  
 {  
 *CPU\_ID* = *CPU\_REG\_PREF* + (*CPU\_NUMBER*++);  
 }  
 public CPU(String data)  
 {  
  
 }  
 public CPU(Builder builder) {  
 CPU\_NAME = builder.Name;  
 CPU\_RELEASE\_YEAR = builder.Year;  
 CPU\_CORES = builder.Core;  
 CPU\_BASE\_MIN = builder.Base\_Min;  
 CPU\_BASE\_MAX = builder.Base\_Max;  
 CPU\_PRICE = builder.Price;  
 // validate();  
 }  
 public CPU(String cpu\_name, int cpu\_release\_date, int cpu\_core, double cpu\_base\_min, double cpu\_base\_max, double cpu\_price)  
 {  
 CPU\_NAME = cpu\_name;  
 CPU\_RELEASE\_YEAR = cpu\_release\_date;  
 CPU\_CORES = cpu\_core;  
 CPU\_BASE\_MIN = cpu\_base\_min;  
 CPU\_PRICE = cpu\_price;  
 CPU\_BASE\_MAX = cpu\_base\_max;  
 }  
 public boolean validateYear(int minRelease, int maxRelease) {  
 if (CPU\_RELEASE\_YEAR < minRelease || CPU\_RELEASE\_YEAR > maxRelease) {  
 return false;  
 }  
 return true;  
 }  
 public boolean validatePrice(int minPrice, int maxPrice)  
 {  
 if (CPU\_PRICE < minPrice || CPU\_PRICE > maxPrice) {  
 return false;  
 }  
 return true;  
 }  
 @Override  
 public void parse(String dataString) {  
 try{  
 Scanner p = new Scanner(dataString).useDelimiter(",");  
 CPU\_NAME = p.next();  
 CPU\_RELEASE\_YEAR = p.nextInt();  
 CPU\_CORES = p.nextInt();  
 CPU\_BASE\_MIN = Double.*parseDouble*(p.next());  
 CPU\_BASE\_MAX = Double.*parseDouble*(p.next());  
 CPU\_PRICE = Double.*parseDouble*(p.next());  
 }  
 catch (NoSuchElementException e)  
 {  
 Ks.*ern*("Not enough data about CPU" + dataString);  
 }  
 }  
 @Override  
 public int hashCode() {  
 int hash = 5;  
 hash = 29 \* hash + Objects.*hashCode*(this.CPU\_NAME);  
 hash = 29 \* hash + this.CPU\_RELEASE\_YEAR;  
 hash = 29 \* hash + this.CPU\_CORES;  
 hash = 29 \* hash + (int) (Double.*doubleToLongBits*(this.CPU\_BASE\_MIN));  
 hash = 29 \* hash + (int) (Double.*doubleToLongBits*(this.CPU\_BASE\_MAX));  
 hash = 29 \* hash + (int) (Double.*doubleToLongBits*(this.CPU\_PRICE) ^ (Double.*doubleToLongBits*(this.CPU\_PRICE) >>> 32));  
 return hash;  
 }  
 @Override  
 public boolean equals(Object obj) {  
 if (this == obj) {  
 return true;  
 }  
 if (obj == null) {  
 return false;  
 }  
 if (getClass() != obj.getClass()) {  
 return false;  
 }  
 final CPU other = (CPU) obj;  
 if (this.CPU\_RELEASE\_YEAR != other.CPU\_RELEASE\_YEAR) {  
 return false;  
 }  
 if (this.CPU\_CORES != other.CPU\_CORES) {  
 return false;  
 }  
 if (Double.*doubleToLongBits*(this.CPU\_PRICE) != Double.*doubleToLongBits*(other.CPU\_PRICE)) {  
 return false;  
 }  
 if (Double.*doubleToLongBits*(this.CPU\_BASE\_MAX) != Double.*doubleToLongBits*(other.CPU\_BASE\_MAX)) {  
 return false;  
 }  
 if (Double.*doubleToLongBits*(this.CPU\_BASE\_MIN) != Double.*doubleToLongBits*(other.CPU\_BASE\_MIN)) {  
 return false;  
 }  
 if (!Objects.*equals*(this.CPU\_NAME, other.CPU\_NAME)) {  
 return false;  
 }  
 return true;  
 }  
 /\*  
 @Override  
 public int compareTo(CPU car) {  
 return Get\_Cpu\_Id().compareTo(car.Get\_Cpu\_Id());  
 }  
 \*/  
 @Override  
 public String toString()  
 {  
 return String.*format*("%s %d %d %.2f %.2f %.2f", CPU\_NAME, CPU\_RELEASE\_YEAR, CPU\_CORES, CPU\_BASE\_MIN,CPU\_BASE\_MAX, CPU\_PRICE);  
 }  
 /\*  
 public final static Comparator byPrice = (obj1, obj2) -> {  
 double price1 = ((CPU) obj1).Get\_Cpu\_Price();  
 double price2 = ((CPU) obj2).Get\_Cpu\_Price();  
  
 double new\_price = Math.sqrt(price1);  
 double new\_price1 = Math.sqrt(price2);  
 if (new\_price < new\_price1) {  
 return -1;  
 }  
 if (new\_price > new\_price1) {  
 return 1;  
 }  
 return 0;  
 };  
 \*/  
 public String Get\_Cpu\_Name(){return CPU\_NAME;}  
 public int Get\_Cpu\_Release\_Year() {return CPU\_RELEASE\_YEAR;}  
 public int Get\_Cpu\_Cores(){return CPU\_CORES;}  
 public double Get\_Cpu\_Base\_Min(){return CPU\_BASE\_MIN;}  
 public double Get\_Cpu\_Price(){return CPU\_PRICE;}  
 public double Get\_Cpu\_Base\_Max(){return CPU\_BASE\_MAX;}  
 public String Get\_Cpu\_Id() { return *CPU\_ID*;};  
 public static class Builder {  
  
 private final static Random *RANDOM* = new Random(1949); // Atsitiktinių generatorius  
 private final static String[][] *MODELS* = { // galimų automobilių markių ir jų modelių masyvas  
 {"Intel", "Quad-Core", "I3", "I5", "I7"},  
 {"AMD", "Ahtlon", "Razen"}  
 };  
 private String Name = "";  
 private int Year = -1;  
 private int Core = -1;  
 private double Base\_Min = -1.0;  
 private double Base\_Max = -1.0;  
 private double Price = -1.0;  
  
 public CPU build() {  
 return new CPU(this);  
 }  
  
 public CPU buildRandom() {  
 int ma = *RANDOM*.nextInt(*MODELS*.length); // markės indeksas 0..  
 int mo = 1 + *RANDOM*.nextInt(*MODELS*.length - 1);  
 int serie = 100 + *RANDOM*.nextInt(9000);  
 return new CPU(*MODELS*[ma][0] + " " + *MODELS*[ma][mo] + '-' + serie,  
 1990 + *RANDOM*.nextInt(25),// metai tarp 1990 ir 2009  
 1 + *RANDOM*.nextInt(8),// rida tarp 6000 ir 228000  
 2.5 + *RANDOM*.nextDouble() \* 4.5,  
 4.5 + *RANDOM*.nextDouble() \* 6.0,  
 200 + *RANDOM*.nextDouble() \* 2000);// kaina tarp 800 ir 88800  
 }  
  
 public Builder Year(int year) {  
 this.Year = year;  
 return this;  
 }  
  
 public Builder model(String model) {  
 this.Name = model;  
 return this;  
 }  
 public Builder Core(int core)  
 {  
 this.Core = core;  
 return this;  
 }  
 public Builder Base\_Min(double base)  
 {  
 this.Base\_Min = base;  
 return this;  
 }  
 public Builder Base\_Max(double base)  
 {  
 this.Base\_Max = base;  
 return this;  
 }  
 public Builder price(double price) {  
 this.Price = price;  
 return this;  
 }  
 }  
  
}

**CPU\_DATA.java**

package edu.ktu.ds.lab3.cpu;  
  
import edu.ktu.ds.lab3.utils.HashType;  
import edu.ktu.ds.lab3.utils.ParsableHashMap;  
import edu.ktu.ds.lab3.utils.ParsableMap;  
  
  
  
public class CPU\_Data {  
 ParsableMap<String, CPU> cpuMap= new ParsableHashMap<>(String::new, CPU::new, HashType.*DIVISION*);  
 ParsableMap<String , CPU> cpuMap1= new ParsableHashMap<>(String::new, CPU::new, HashType.*DIVISION*);  
 HashMapOa<String, CPU> test = new HashMapOa<>();  
 String CPU\_Keys[] ={  
 "CPU512", "CPU105","CPU130","CPU230"  
 };  
  
 public void CPU\_Data(boolean Parse)  
 {  
 if(Parse)  
 {  
 CPU CPU4 = new CPU();  
 CPU CPU5 = new CPU();  
 CPU CPU6 = new CPU();  
 CPU CPU7 = new CPU();  
 CPU4.parse("Intel Core i7-8700,2017,6,3.2,4.6,304.99");  
 CPU5.parse("Intel Core i7-9700K,2019,8,3.3,4.9,370.99");  
 CPU6.parse("Intel Core i9-9900K,2018,8,3.6,5.0,484.99");  
 CPU7.parse("AMD RYZEN 9,2019,12,3.8,4.6,499.99");  
 CPU[] cpus ={  
 CPU4, CPU5, CPU6, CPU7  
 };  
 for(int i =0; i < CPU\_Keys.length; i++) {  
 test.put(CPU\_Keys[i], cpus[i]);  
 }  
 }  
 else  
 {  
 CPU Cpu1 = new CPU("AMD RYZEN 9", 2019, 12, 3.8,4.6, 499.99);  
 CPU Cpu2 = new CPU("Intel Core i9-9900K", 2018, 8, 3.6, 5.0, 484.99);  
 CPU Cpu8= new CPU("Intel Core i7-9700K", 2019, 8, 3.3, 4.9, 370.99);  
 CPU Cpu3= new CPU("Intel Core i5-5800", 2015, 4, 2.5, 3.5, 250.99);  
 CPU[] cpus ={  
 Cpu1, Cpu2, Cpu8,Cpu3  
 };  
 for(int i =0; i < CPU\_Keys.length; i++) {  
 test.put(CPU\_Keys[i], cpus[i]);  
 }  
 }  
 }  
}

**HashMapOa.java – Kvadratinis dėstymas**

package edu.ktu.ds.lab3.cpu;  
  
import edu.ktu.ds.lab3.utils.HashType;  
import edu.ktu.ds.lab3.utils.Ks;  
import edu.ktu.ds.lab3.utils.Map;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class HashMapOa<K, V> implements Map<K, V> {  
 public static final int *DEFAULT\_INITIAL\_CAPACITY* = 16;  
 public static final float *DEFAULT\_LOAD\_FACTOR* = 0.9f;  
 public static final HashType *DEFAULT\_HASH\_TYPE* = HashType.*DIVISION*;  
 protected HashType ht;  
  
 // Maišos lentelė  
 protected Node<K, V>[] table;  
 protected int size = 0;  
 protected float loadFactor;  
 protected int rehashesCounter = 0;  
 protected int index = 0;  
  
 public HashMapOa()  
 {  
 ht = *DEFAULT\_HASH\_TYPE*;  
 this.table = new Node[*DEFAULT\_INITIAL\_CAPACITY*];  
 this.loadFactor = *DEFAULT\_LOAD\_FACTOR*;  
 }  
 public HashMapOa(int initialCapacity, HashType ht, float loadFactor) {  
 if (initialCapacity <= 0) {  
 throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);  
 }  
 this.table = new Node[initialCapacity];  
 this.loadFactor = loadFactor;  
 this.ht = ht;  
 }  
 @Override  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 @Override  
 public int size() {  
 return size;  
 }  
 public int getTableCapacity() {  
 return table.length;  
 }  
 @Override  
 public void clear() {  
 Arrays.*fill*(table, null);  
 size = 0;  
 index = 0;  
 rehashesCounter = 0;  
 }  
  
 @Override  
 public String[][] toArray() {  
 String[][] result = new String[table.length][];  
 int count = 0;  
 for (Node<K, V> n : table) {  
 String[] list = new String[1];  
 list[0] = n.toString();  
 result[count] = list;  
 count++;  
 }  
 return result;  
 }  
 @Override  
 public V put(K key, V value) {  
 if (key == null || value == null) {  
 throw new IllegalArgumentException("Key or value is null in put(Key key, Value value)");  
 }  
 index = hash(key, ht);  
 int inde = FindPosition(key);  
 table[inde] = new Node<>(key, value, table[inde]);  
 size++;  
 if (size >= table.length \* loadFactor) {  
 rehash();  
 }  
 return value;  
 }  
  
 @Override  
 public V get(K key) {  
 if (key == null) {  
 throw new IllegalArgumentException("Key is null in get(Key key)");  
 }  
 index = hash(key, ht);  
 int index1 = GetIndex(key);  
 if(index1 >= 0)  
 {  
 return table[index].value;  
 }  
 return null;  
 }  
 private int FindPosition(K key)  
 {  
 index = hash(key, ht);  
 int index0 = index;  
 for(int i =0; i < table.length; i++)  
 {  
 if(table[index] == null || table[index].key.equals(key))  
 {  
 return index;  
 }  
 index = (index + (i + 1) \* (i + 1)) % table.length;  
 }  
 return -1;  
 }  
 public int GetIndex(K key)  
 {  
 index = hash(key, ht);  
 int index0 = index;  
 for(int i =0; i < table.length; i++)  
 {  
 if(table[index].key != null) {  
 if (table[index].key.equals(key)) {  
 return index;  
 }  
 }  
 index = (index + (i + 1) \* (i + 1)) % table.length;  
 }  
 return -1;  
 }  
 public int getRehashesCounter() {  
 return rehashesCounter;  
 }  
 @Override  
 public V remove(K key) {  
 if (key == null) {  
 throw new IllegalArgumentException("Key is null in remove(Key key)");  
 }  
 int index1 = GetIndex(key);  
 if(index1 > -1) {  
 V value = table[index1].value;  
 table[index1].key = null;  
 table[index1].value = null;  
 size--;  
 return value;  
 }  
 return null;  
 }  
  
 @Override  
 public boolean contains(K key) {  
 return get(key) != null;  
 }  
 private int hash(K key, HashType hashType) {  
 int h = key.hashCode();  
 switch (hashType) {  
 case *DIVISION*:  
 return Math.*abs*(h) % table.length;  
 case *MULTIPLICATION*:  
 double k = (Math.*sqrt*(5) - 1) / 2;  
 return (int) (((k \* Math.*abs*(h)) % 1) \* table.length);  
 case *JCF7*:  
 h ^= (h >>> 20) ^ (h >>> 12);  
 h = h ^ (h >>> 7) ^ (h >>> 4);  
 return h & (table.length - 1);  
 case *JCF8*:  
 h = h ^ (h >>> 16);  
 return h & (table.length - 1);  
 default:  
 return Math.*abs*(h) % table.length;  
 }  
 }  
 public void println() {  
 if (isEmpty()) {  
 Ks.*oun*("Atvaizdis yra tuščias");  
 } else {  
 String[][] data = getModelList();  
 for (int i = 0; i < data.length; i++) {  
 for(int j =0; j < data[i].length; j++) {  
 String format = "%7s";  
 Object value = data[i][j];  
 Ks.*ouf*(format, (value == null ? "" : value));  
 }  
 Ks.*oufln*("");  
 }  
 }  
  
 Ks.*oufln*("\*\*\*\*\*\* Bendras porų kiekis yra " + size);  
 }  
 public String[][] getModelList() {  
 String[][] result = new String[table.length][];  
 int count = 0;  
 for (Node<K, V> n : table) {  
 List<String> list = new ArrayList<>();  
 list.add("[ " + count + " ]");  
 if(n != null) {  
 list.add("-->");  
 list.add(n.key.toString());  
 }  
 result[count] = list.toArray(new String[0]);  
 count++;  
 }  
 return result;  
 }  
 private void rehash() {  
 HashMapOa<K, V> newMap = new HashMapOa<>(table.length \* 2, ht, loadFactor);  
 for (int i = 0; i < table.length; i++) {  
 if(table[i] != null) {  
 newMap.put(table[i].key, table[i].value);  
 }  
 }  
 table = newMap.table;  
 rehashesCounter++;  
 }  
 protected static class Node<K, V> {  
  
 // Raktas  
 protected K key;  
 // Reikšmė  
 protected V value;  
  
 protected Node() {  
 }  
  
 protected Node(K key, V value, Node<K, V> next) {  
 this.key = key;  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return key + "=" + value;  
 }  
 }  
  
}

**HashMap.java – Pridėti metodai**

public List<V> values()  
{  
 if(size == 0)  
 {  
 return null;  
 }  
 List<V> list = new ArrayList<>();  
 for(int index =0; index < table.length; index++) {  
 for (Node<K, V> n = table[index]; n != null; n = n.next) {  
 list.add(n.value);  
 }  
 }  
 return list;  
}

public Set<K> keySet()  
{  
 if(size == 0)  
 {  
 return null;  
 }  
 Set<K> set = new HashSet<>();  
 for(int index =0; index < table.length; index++) {  
 for (Node<K, V> n = table[index]; n != null; n = n.next) {  
 set.add(n.key);  
 }  
 }  
 return set;  
}

public V putIfAbsent(K key, V value)  
{  
 if (key == null) {  
 throw new IllegalArgumentException("Key is null");  
 }  
 index = hash(key, ht);  
 if(table[index] == null)  
 {  
 put(key, value);  
 return null;  
 }  
 else  
 {  
 return table[index].value;  
 }  
}  
public boolean containsValue(V value)  
{  
 if (value == null) {  
 throw new IllegalArgumentException("Value is null");  
 }  
 for(int index =0; index < table.length; index++) {  
 for (Node<K, V> n = table[index]; n != null; n = n.next) {  
 if (n.value == value) {  
 return true;  
 }  
 }  
 }  
 return false;  
}  
public int numberOfEmpties()  
{  
 int emptyTableCell = 0;  
 for(int index =0; index < table.length; index++) {  
 if(table[index] == null){  
 emptyTableCell++;  
 }  
 }  
 return emptyTableCell;  
}

**JUNIT.java – Nėra pagrindiniu metodų testavimai**

package edu.ktu.ds.lab3.cpu;  
  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.DisplayName;  
import org.junit.jupiter.api.Test;  
  
import java.util.Random;  
  
import static org.junit.jupiter.api.Assertions.*assertTrue*;  
  
public class JUNIT {  
  
 static CPU\_Generator *generator* = new CPU\_Generator();  
 static CPU[] *CPU\_List*;  
 static CPU *cpu* = new CPU();  
 @BeforeAll  
 public static void Create()  
 {  
 Random rand = new Random();  
 *CPU\_List* = *generator*.generateShuffleCpuAndKeys(50, 10);  
 *cpu* = *CPU\_List*[rand.nextInt(*CPU\_List*.length - 1)];  
 }  
 @DisplayName("Checking Year")  
 @Test  
 public void Check\_Year()  
 {  
 *assertTrue*(*cpu*.validateYear(2019, 2020), String.*format*("Error year is too low or too high %d", *CPU\_List*[0].Get\_Cpu\_Release\_Year()));  
 }  
 @DisplayName("Checking Price")  
 @Test  
 public void Check\_Price()  
 {  
 *assertTrue*(*cpu*.validatePrice(2500, 3000), String.*format*("Error price is out of bound %.2f", *CPU\_List*[0].Get\_Cpu\_Price()));  
 }  
}

**JAVAFX:**

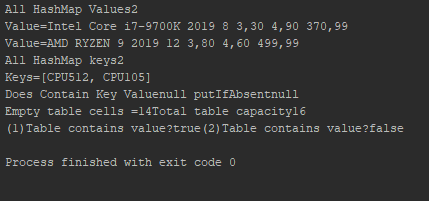


**Data:**

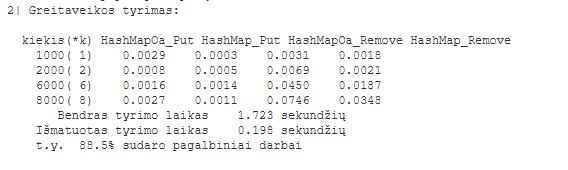
String CPU\_Keys[] ={  
 "CPU512", "CPU105","CPU130","CPU230"  
};  
  
CPU Cpu1 = new CPU("AMD RYZEN 9", 2019, 12, 3.8,4.6, 499.99);

CPU Cpu2 = new CPU("Intel Core i9-9900K", 2018, 8, 3.6, 5.0, 484.99);  
CPU Cpu3 = new CPU("Intel Core i7-9700K", 2019, 8, 3.3, 4.9, 370.99);

CPU CPU4 = new CPU();  
CPU4.parse("Intel Core i7-8700K,2017,6,3.7,4.7,354.99");



**Greitaveikos rezultatai:**



Išvados

Kvadratinis dėstymas mano atveju yra per puse lėtesnis nei grandinėlimis, paieška yra greitesnė grandinėlėse todėl ir galime atlikti greitesni idėjimą arba ištrinimą.

Aišku kvadratiniam dėstyme galime pasiekti geresniu rezultatu nei grandinėlese jeigu nebūtų susidurimu(collision).