

**Kauno technologijos universitetas**

Informatikos fakultetas

HashMap

P175B014 Duomenų struktūrų trečias laboratorinis darbas

**Projekto autorius**

Gustas Klevinskas

**Akademinė grupė**

IFF-8/7

Kaunas, 2019

Turinys

[HashMap metodai 3](#_Toc26395958)

[containsValue(K key) 3](#_Toc26395959)

[putIfAbsent(K key, V value) 4](#_Toc26395960)

[replace(K key, V oldValue, V newValue) 5](#_Toc26395961)

[replaceAll(V oldValue, V newValue) 6](#_Toc26395962)

[numberOfEmpties() 7](#_Toc26395963)

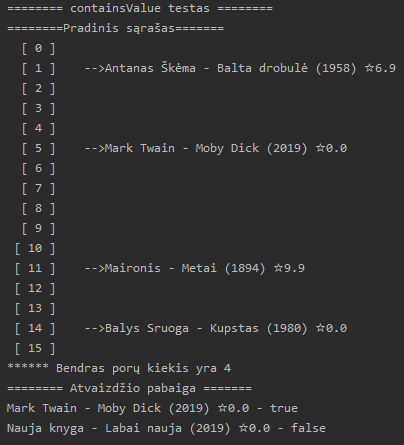
[HashMapOa 8](#_Toc26395964)

[Greitaveika 12](#_Toc26395965)

# HashMap metodai

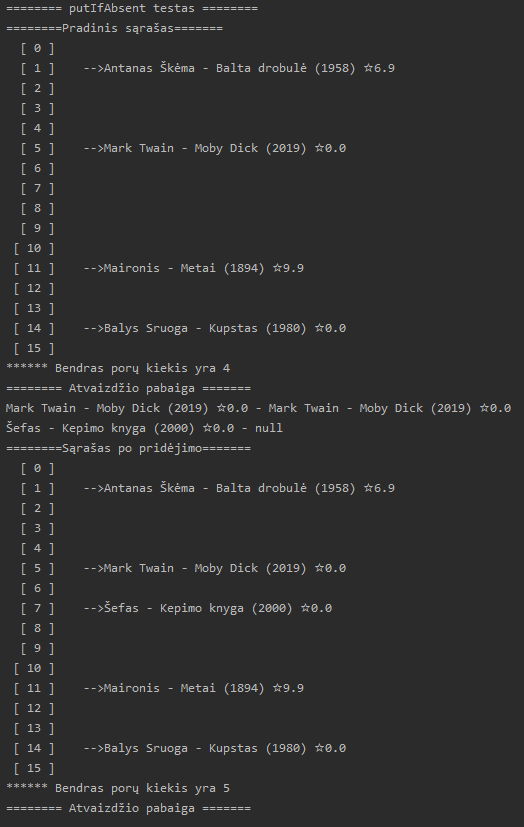
## containsValue(K key)

public boolean containsValue(Object value) {  
 return toString().contains(value.toString());  
}



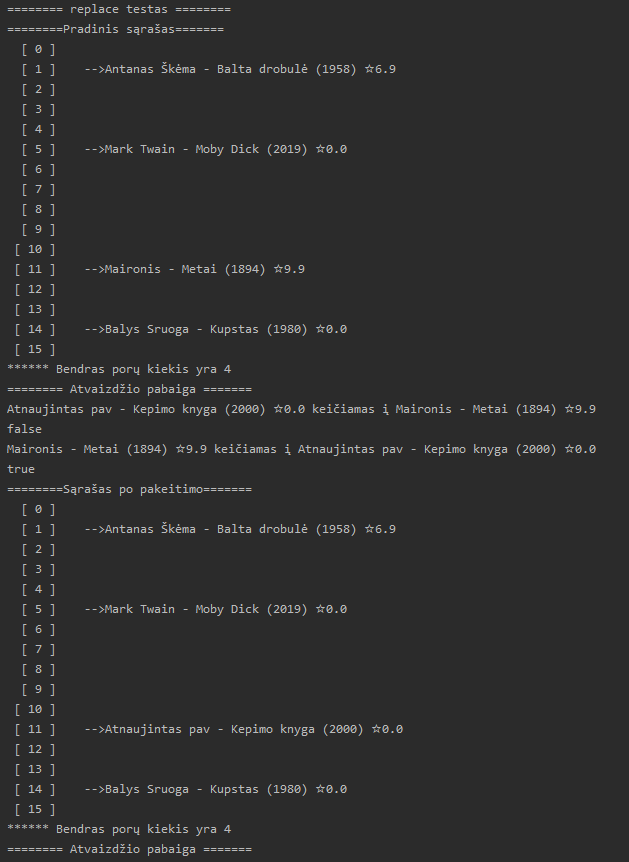
## putIfAbsent(K key, V value)

public V putIfAbsent(K key, V value) {  
 if (get(key) == null) {  
 put(key, value);  
 return null;  
 }  
  
 return value;  
}



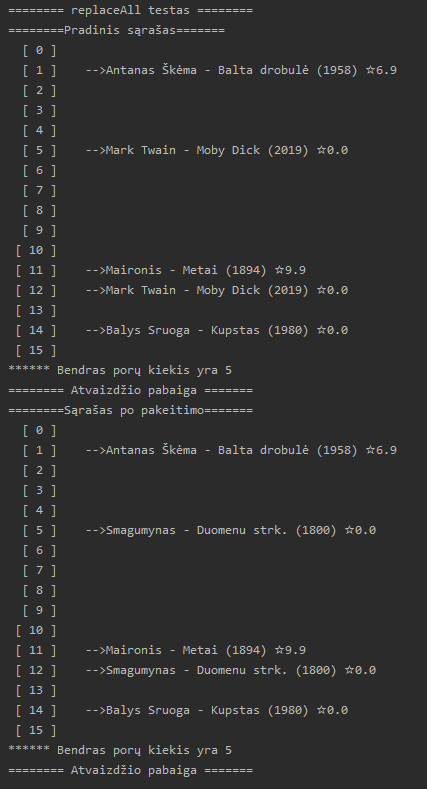
## replace(K key, V oldValue, V newValue)

public boolean replace(K key, V oldValue, V newValue) {  
 if (get(key) != null && get(key).equals(oldValue)) {  
 Node<K, V> node = getInChain(key, table[hash(key, ht)]);  
 node.value = newValue;  
 return true;  
 }  
  
 return false;  
}



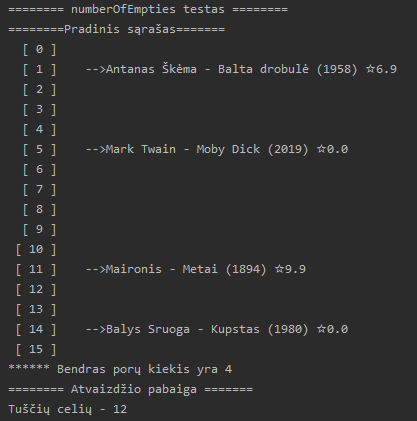
## replaceAll(V oldValue, V newValue)

public void replaceAll(V oldValue, V newValue) {  
 for (Node<K, V> i : table) {  
 for (Node<K, V> n = i; n != null; n = n.next) {  
 if (n.value.equals(oldValue))  
 n.value = newValue;  
 }  
 }  
}



## numberOfEmpties()

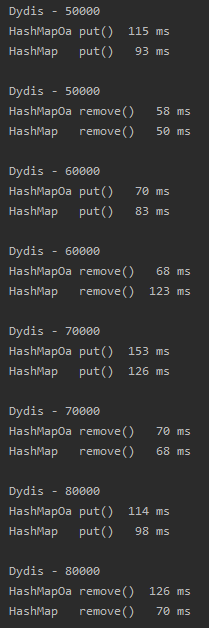
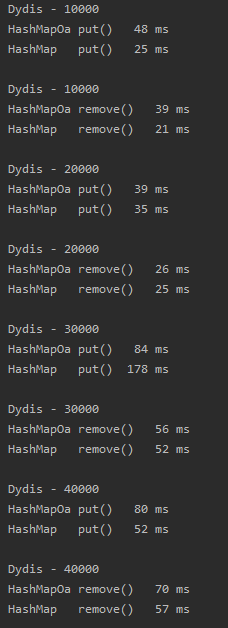
public int numberOfEmpties() {  
 int emptyCells = 0;  
  
 for (Node<K, V> i : table) {  
 if (i == null)  
 emptyCells++;  
 }  
  
 return emptyCells;  
}

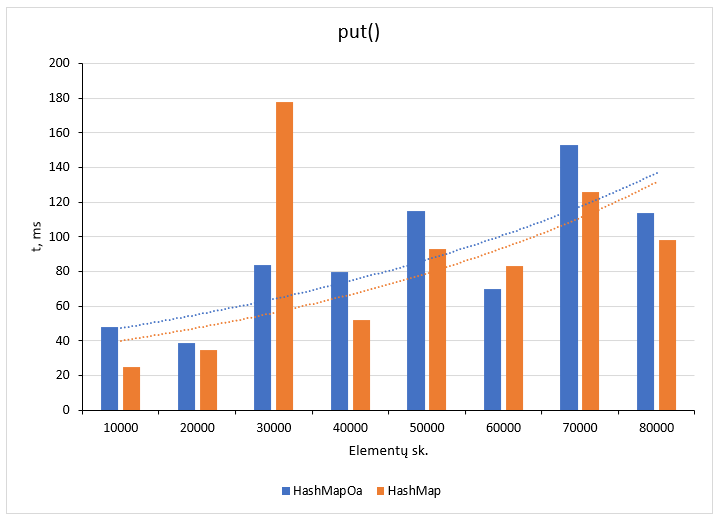


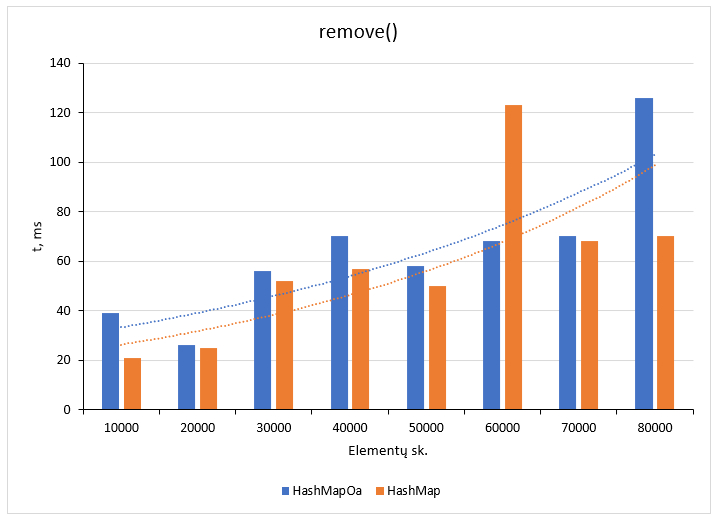
# HashMapOa

package edu.ktu.ds.lab3.klevinskas;  
  
import edu.ktu.ds.lab3.utils.HashType;  
import edu.ktu.ds.lab3.utils.Map;  
  
import java.util.Arrays;  
  
public class HashMapOa<K, V> implements Map<K, V> {  
  
 private static final int *DEFAULT\_INITIAL\_CAPACITY* = 16;  
 private static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  
 private static final HashType *DEFAULT\_HASH\_TYPE* = HashType.*DIVISION*;  
  
 private Entry<K, V>[] table;  
 private int size = 0;  
 private int deletedEntries = 0;  
 private float loadFactor;  
 private HashType ht;  
  
 public HashMapOa() {  
 this(*DEFAULT\_HASH\_TYPE*);  
 }  
  
 public HashMapOa(HashType ht) {  
 this(*DEFAULT\_INITIAL\_CAPACITY*, ht);  
 }  
  
 public HashMapOa(int initialCapacity, HashType ht) {  
 this(initialCapacity, *DEFAULT\_LOAD\_FACTOR*, ht);  
 }  
  
 public HashMapOa(float loadFactor, HashType ht) {  
 this(*DEFAULT\_INITIAL\_CAPACITY*, loadFactor, ht);  
 }  
  
 public HashMapOa(int initialCapacity, float loadFactor, HashType ht) {  
 if (initialCapacity <= 0) {  
 throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);  
 }  
  
 if ((loadFactor <= 0.0) || (loadFactor > 1.0)) {  
 throw new IllegalArgumentException("Illegal load factor: " + loadFactor);  
 }  
  
 this.table = new Entry[initialCapacity];  
 this.loadFactor = loadFactor;  
 this.ht = ht;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 @Override  
 public int size() {  
 return size;  
 }  
  
 @Override  
 public void clear() {  
 Arrays.*fill*(table, null);  
 size = 0;  
 }  
  
 @Override  
 public String[][] toArray() {  
 String[][] result = new String[table.length][];  
  
 for (int i = 0; i < table.length; i++) {  
 result[i] = new String[]{table[i].toString()};  
 }  
  
 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(K key, V value)");  
  
 table[getUnusedIndex(key)] = new Entry<>(key, value);  
 size++;  
  
 if (size + deletedEntries > table.length \* loadFactor)  
 rehash();  
  
 return value;  
 }  
  
 @Override  
 public V get(K key) {  
 if (key == null)  
 throw new IllegalArgumentException("Key is null in get(K key)");  
  
 if (getIndex(key) != -1)  
 return table[getIndex(key)].value;  
 else  
 return null;  
 }  
  
 @Override  
 public V remove(K key) {  
 V removedValue = get(key);  
  
 if (removedValue != null) {  
 table[getIndex(key)] = new Entry<>(null, null);  
 size--;  
 deletedEntries++;  
 }  
  
 return removedValue;  
 }  
  
 @Override  
 public boolean contains(K key) {  
 return get(key) != null;  
 }  
  
 private int getIndex(K key) {  
 int index = hash(key, ht);  
  
 for (int i = 0; i < table.length; i++) {  
 int tempIndex = (index + i) % table.length;  
 Entry<K, V> tempEntry = table[tempIndex];  
  
 if (tempEntry != null && tempEntry.key != null && tempEntry.key.equals(key)) {  
 return tempIndex;  
 }  
 }  
  
 return -1;  
 }  
  
 private int getUnusedIndex(K key) {  
 int index = hash(key, ht);  
  
 for (int i = 0; i < table.length; i++) {  
 int tempIndex = (index + i) % table.length;  
 Entry<K, V> tempEntry = table[tempIndex];  
  
 if (tempEntry == null || tempEntry.key == null)  
 return tempIndex;  
 }  
  
 return -1;  
 }  
  
 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;  
 }  
 }  
  
 private void rehash() {  
 HashMapOa<K, V> newMap = new HashMapOa<>(table.length \* 2, loadFactor, ht);  
  
 for (Entry<K, V> i : table)  
 if (i != null && i.key != null)  
 newMap.put(i.key, i.value);  
  
 table = newMap.table;  
 deletedEntries = 0;  
 }  
  
 private static class Entry<K, V> {  
  
 protected K key;  
 protected V value;  
  
 protected Entry() {  
  
 }  
  
 protected Entry(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return key + "=" + value;  
 }  
 }  
}

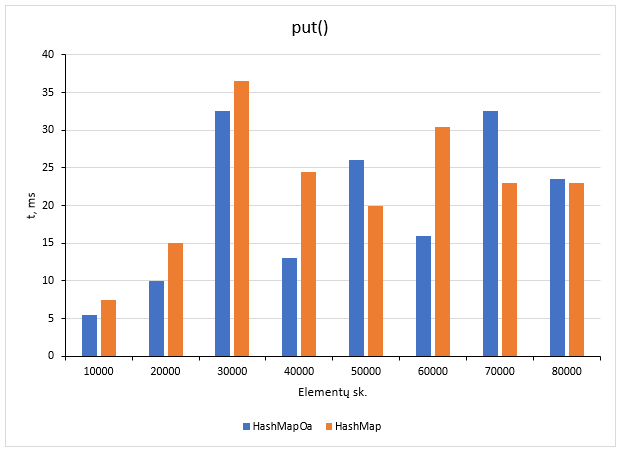
# Greitaveika

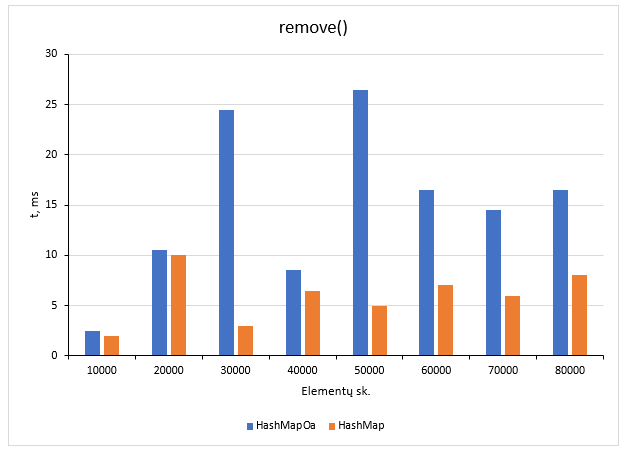






Matome, kad ir įdėjimo, ir pašalinimo metodai vidutiniškai užtrunka ilgiau naudojant HashMapOa nei HashMap. Skaičiuojant sugaištą laiką atliekant veiksmus pasitaikė kelios reikšmės, kurios stipriai nukrypusios nuo vidurkio. Tai galėjo nutikti, nes įrašinėjant (ir ištrinant) reikšmes, jos buvo skaitomos iš failo. Pamėginau surašyti visus žodžius į masyvą prieš atliekant pridėjimą ir šalinimą iš HashMap‘ų ir gavau tokius rezultatus:





Matome, kad vis tiek išlieka kelios reikšmės, kurios yra nukrypusios nuo vidutinės reikšmės. Iš antrų grafikų matome, kad didėjant elementų kiekiui, atsiranda tendencija ilgėti operacijoms. Tai tikriausiai atsitinka dėl to, kad didėja kolizijų kiekis ir operacijos nebeatliekamos per konstantinį laiką.

Šalinimo operacija vidutiniškai užtrunka ilgiau HashMapOa struktūroje nei HashMap, kur naudojama grandinėlės kolizijų sprendimo būdas. Tuo tarpu pridėjimo operacija atliekama greičiau HashMapOa struktūroje, tačiau skirtumas yra ganėtinai smulkus (<5 ms), tad tikėtina, kad atsirado kitų veiksnių, lėtinančių veikimą.