Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЛАБОРАТОРНА РОБОТА № 5

з дисципліни «Теорія алгоритмів»

на тему «Хеш-таблиці»

ВИКОНАВ:
студент 4 курсу
групи ІП-72з
Сахнюк Антон Юрійович
Залікова - 6224

ПЕРЕВІРИВ:
Доцент кафедри ОТ
к.т.н., с.н.с.
Антонюк А.І.

Київ - 2021

ЗАВДАННЯ

Тема: “Хеш-таблиці”
Мета: Хеш-таблиці (hash-tables) можуть використовуватись для збереження масивів даних, швидкого доступу, вставки та видалення елементів. За допомогою хеш-таблиць можна ефективно розв'язати наступну задачу. Нехай заданий масив чисел A та число S. Потрібно дізнатись, чи присутні в масиві A два числа, сума яких дорівнює S.
Завдання: В роботі необхідно реалізувати різні типи хеш-таблиць із використанням різних хеш-функцій для розв’язання наведеної вище задачі. При цьому потрібно порівняти ефективність різних підходів шляхом підрахунку кількості колізій для кожного типу хеш-функцій та хеш-таблиць.

ПРОГРАМНИЙ КОД

Файл Map.kt:

```
package ua.santoni7.l5


import kotlin.collections.List


/**
 * Інтерфейс що інкапсулює хеш-таблицю. Підтримує вставку пари ключ-значення і пошук значення по ключу
 *
 * Див. реалізації [ChainingHashMap] та [OpenAddressHashMap]
 */
interface Map<K, V> {


    fun getEntries(): List<KeyValuePair<K, V>>


    operator fun set(key: K, value: V): Boolean


    operator fun get(key: K): KeyValuePair<K, V>?


    fun size(): Int


    fun countCollisions(): Int
}
```

```
class KeyValuePair<K, V>(var key: K, var value: V) {
    override fun toString() = "{ $key = > $value }"
}
```

Файл ChainingHashMap.kt:

```
class ChainingHashMap<K, V>(
    private val hashProvider: HashProvider<K> = HashProvider.createDefault(),
    initCapacity: Int = 1024
) : Map<K, V> {
    private var capacity = initCapacity
    private var size = 0

    //Each element is a head of linked list of Entries
    private var mEntries: Array<Entry?>

    init {
        mEntries = Array(capacity) { null }
    }

    // Double capacity and reinsert all entries at new positions
    private fun enlarge() {
        val oldEntries = mEntries
        size = 0
        capacity *= 2

        mEntries = Array(capacity) { null }
        for (i in 0 until capacity / 2) {
            var e: Entry? = oldEntries[i]
            while (e != null) {
                set(e.keyValuePair)
                e = e.next()
            }
        }
    }
```

```
    /**
     * Calculate index in Entries array based on item's hash provided by
[hashProvider]
     */
    private fun indexFor(item: K, capacity: Int): Int {
        return hashProvider.hashFor(item, capacity)
    }

    override fun set(key: K, value: V): Boolean {
        return set(KeyValuePair(key, value))
    }

    private fun set(pair: KeyValuePair<K, V>): Boolean {
        if (thresholdSize())
            enlarge()

        val index = indexFor(pair.key, capacity)
        val entry = mEntries[index]
        if (entry == null) {
            mEntries[index] = Entry(pair)
        } else {
            val kvp = entry.find(pair.key)
            if (kvp == null) {
                //Key is not in the table
                entry.append(pair)
            } else {
                // Key already exists, so change value
                kvp.value = pair.value
            }
        }
        return true
    }

    override fun get(key: K): KeyValuePair<K, V>? {
        val index = indexFor(key, capacity)
        return mEntries[index]?.find(key)
```

```
}

override fun size(): Int {
    return size
}

override fun getEntries(): List<KeyValuePair<K, V>> {
    val list = mutableListOf<KeyValuePair<K, V>>()
    for (i in 0 until capacity) {
        var e = mEntries[i]
        while (e != null) {
            list.add(e.keyValuePair)
            e = e.next()
        }
    }
    return list
}

override fun countCollisions(): Int {
    var c = 0
    for (i in 0 until capacity) {
        val e = mEntries[i]
        c += e?.sizeToEnd()?.minus(1) ?: 0
    }
    return c
}

// Indicates whether capacity should be enlarged
private fun thresholdSize(): Boolean {
    return size >= capacity * loadFactor
}

internal inner class Entry(var keyValuePair: KeyValuePair<K, V>) {
    var nextEntry: Entry? = null

    operator fun next(): Entry? {
```

```
            return nextEntry
        }

        // Append to list
        fun append(pair: KeyValuePair<K, V>) {
            if (nextEntry == null) {
                nextEntry = Entry(pair)
            } else {
                nextEntry!!.append(pair)
            }
        }

        // Recursively find key in list
        fun find(key: K): KeyValuePair<K, V>? {
            if (keyValuePair.key?.equals(key) == true)
                return keyValuePair
            return if (next() != null) next()!!.find(key) else null
        }

        fun sizeToEnd(): Int {
            var c = 1
            var entry: Entry? = this
            while(entry?.next() != null){
                entry = entry.next()
                c++
            }
            return c
        }
    }

    companion object {
        private val loadFactor = 0.9999f
    }

}
```

Файл OpenAddressHashMap.kt:

```kotlin
package ua.santoni7.l5


/**
 * HashTable implementation based on open address
 */
class OpenAddressHashMap<K, V>(
    private val hashProvider: HashProvider<K> = HashProvider.createDefault(),
    initCapacity: Int = 1024
) : Map<K, V> {
    private var capacity = initCapacity
    private var size = 0

    //Each element is a head of linked list of Entries
    private var mEntries: Array<Entry?>

    init {
        mEntries = Array(capacity) { null }
    }

    // Double capacity and reinsert all entries at new positions
    private fun enlarge() {
        val oldEntries = mEntries
        size = 0
        capacity *= 2

        mEntries = Array(capacity) { null }
        for (i in 0 until capacity / 2) {
            oldEntries[i]?.let { set(it.keyValuePair) }
        }
    }

    /**
     * Calculate index in Entries array based on item's hash provided by [hashProvider]
     */
```

```
    private fun indexFor(item: K, capacity: Int): Int {
        return hashProvider.hashFor(item, capacity)
    }

    override fun set(key: K, value: V): Boolean {
        return set(KeyValuePair(key, value))
    }

    private fun set(pair: KeyValuePair<K, V>): Boolean {
        if (thresholdSize())
            enlarge()

        var index = indexFor(pair.key, capacity)

        while(mEntries[index] != null && mEntries[index]?.isDeleted != true &&
mEntries[index]?.key != pair.key) {
            index = (index + 1) % capacity // TODO: Add probing provider
        }
        mEntries[index] = Entry(pair)
        return true
    }

    override fun get(key: K): KeyValuePair<K, V>? {
        var index = indexFor(key, capacity)
        var c = 0
        while(mEntries[index]?.key != key && c < capacity){
            index = (index + 1) % capacity
            c++
        }
        return if(mEntries[index]?.key == key) mEntries[index]?.keyValuePair
else null
    }

    override fun size(): Int {
        return size
    }
```

```
    override fun getEntries(): List<KeyValuePair<K, V>> {
        val list = mutableListOf<KeyValuePair<K, V>>()
        for (i in 0 until capacity) {
            mEntries[i]?.keyValuePair?.let { list.add(it) }
        }
        return list
    }

    override fun countCollisions(): Int {
        var c = 0
        mEntries.forEachIndexed { index, entry ->
            entry?.key?.let { hashProvider.hashFor(it, capacity) }?.let
{ calculatedIndex ->
                if(index != calculatedIndex) c++ // Collision found: element is
located on position different then hashProvider provided
            }
        }
        return c
    }

    // Indicates whether capacity should be enlarged
    private fun thresholdSize(): Boolean {
        return size >= capacity * loadFactor
    }



    internal inner class Entry(val keyValuePair: KeyValuePair<K, V>) {
        val key get() = keyValuePair.key
        val value get() = keyValuePair.value

        var isDeleted = false
    }

    companion object {
        private val loadFactor = 0.75f
    }
```

```
}
```

Файл HashProvider.kt:

```
package ua.santoni7.l5


import kotlin.math.abs


/**
 * Інтерфейс що інкапсулює обчислення хеш функції і отримання індексу комірки всередині хеш таблиці
 */
interface HashProvider<T> {
    /**
     * Must return an integer hash-function value which is in [0; capacity) range
     */
    fun hashFor(value: T, capacity: Int): Int


    companion object {
        fun <R> create(provider: (value: R, capacity: Int) -> Int): HashProvider<R> = object :
            HashProvider<R> {
            override fun hashFor(value: R, capacity: Int) = provider.invoke(value, capacity)
        }


        fun <R> createDefault(): HashProvider<R> =
            create { value, capacity -> abs(value.hashCode()) and capacity - 1 }
    }
}


/**
 * Варіанти реалізації хеш функцій для цілих чисел
 */
object IntHashProviders {
    val ALL = listOf(Default, PseudoRandom, ModCapacity)


    /**
```

```
     * Використовує вбудовану у JVM імплементацію Object::hashCode(). Може різнитись в залежності від середовища, в даному випадку тести проводились на JDK 11
     */
    object Default :
        HashProvider<Int> by HashProvider.create(provider = { value, capacity -> abs(value.hashCode()) % capacity })

    /**
     * Використовує псевдо-випадкову функцію побудовану на операціях XOR і знакового/беззнакового побітового зсуву:
     */
    object PseudoRandom : HashProvider<Int> by HashProvider.create<Int>(provider = { value, capacity ->
        abs(pseudoRandomHashFunction(value)) and (capacity - 1)
    })

    /**
     * Обраховує хеш як остачу від ділення абсолютного значення ключа на ємнітсть таблиці
     */
    object ModCapacity : HashProvider<Int> by HashProvider.create<Int>(provider = { value, capacity ->
        abs(value) % (capacity - 1)
    })

    // pseudo-random hash function
    private fun pseudoRandomHashFunction(value: Int): Int {
        var a = value
        a = a xor (a shl 13)
        a = a xor a.ushr(17)
        a = a xor (a shl 5)
        return a
    }
}
```

Файл Lab5.kt:

```
package ua.santoni7.l5
```

```
import kotlin.random.Random


/**
 * Пошук двох чисел з масиву [A] що у сумі дають число [s]
 */
fun findTwoSum(A: IntArray, s: Int, hashMap: Map<Int, Int>):
Boolean {
    for (i in 0 until A.size) {
        hashMap[A[i]] = i
    }


    for (i in 0 until A.size) {
        val x = A[i]
        val y = s - x
        val j = hashMap[y]?.value
        if (j != null && i != j) {
            println("Знайдено пару чисел з масиву A що дають у
сумі S: A[$i]+A[$j]=${A[i]}+${A[j]}=$s")
            return true
        }
    }
    return false
}


val initCapacity = 64
fun main() {
    val mapFactories = MapFactory.ALL // усі наявні імплементації
інтерфейсу Map
    val hashProviders = IntHashProviders.ALL // усі наявні
імплементації інтерфейсу HashProvider


    val results = mutableMapOf<MapFactory,
MutableMap<HashProvider<Int>, Int>>() // структура для збереженні
к-сті колізії для кожного варіанту
```

```
    val A = generateArray(25, 0, 25)
    val s = 48

    mapFactories.forEach { factory ->
        val resultsMap = mutableMapOf<HashProvider<Int>, Int>()//
ChainingHashMap<HashProvider<Int>, Int>()
        hashProviders.forEach { hashProvider ->
            val hashMap = factory.createMap(hashProvider)
            findTwoSum(A, s, hashMap)
            resultsMap[hashProvider] = hashMap.countCollisions()
        }
        results[factory] = resultsMap
    }

    results.forEach { it ->
        val mapType = it.key.name
        println("Results for $mapType")
        println(it.value.entries.joinToString(separator = "\n")
{ "\t${it.key::class.simpleName} => ${it.value}" })
    }
}

/**
 * Генерація випадкового масиву
 */
val random = Random(System.currentTimeMillis())

fun generateArray(size: Int, min: Int, max: Int): IntArray =
IntArray(size) { random.nextInt(min, max) }

fun readNumber(prompt: String = "Input number: "): Int {
    print(prompt);
    val s = readLine()
```

```
    return s?.toInt() ?: throw IllegalStateException("Could not
read a number")
}


class MapFactory(
    val name: String,
    private val factory: (HashProvider<Int>) -> Map<Int, Int>
) {
    fun createMap(hashProvider: HashProvider<Int>): Map<Int, Int>
= factory.invoke(hashProvider)
    override fun equals(other: Any?): Boolean {
        return (other as? MapFactory)?.name?.equals(name) ?: false
    }


    override fun hashCode() = name.hashCode()
    override fun toString(): String = name


    companion object {
        val CHAINING = MapFactory("Chaining")
{ ChainingHashMap(it, initCapacity) }
        val OPEN_ADDRESS = MapFactory("OpenAddress")
{ OpenAddressHashMap(it, initCapacity) }


        val ALL = listOf(CHAINING, OPEN_ADDRESS)
    }
}
```

РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМИ

1. Початковий розмір хеш таблиці 1024. Заповнюється 1023 випадково-згенерованих елементи від 0 до 100000. S = 1000. Нижче наведено кількість колізій у кожному варіанті. Chaining - ланцюжковий метод, OpenAdress - метод відкритої адресації. Див. детальний опис досліджуваних хеш-функцій у файлі HashProvider.kt та у висновку

```
Results for Chaining
    Default => 363
    PseudoRandom => 362
    ModCapacity => 394
Results for OpenAddress
    Default => 493
    PseudoRandom => 500
    ModCapacity => 540

Process finished with exit code 0
```

2х таких чисел що дають у сумі S не знайдено

2. Початковий розмір таблиці 64. Заповнюється 13 елементів від 0 до 20. S=10:

```
Знайдено пару чисел з масиву A що дають у сумі S: A[3]+A[8]=9+1=10
Results for Chaining
    Default => 0
    PseudoRandom => 0
    ModCapacity => 0
Results for OpenAddress
    Default => 0
    PseudoRandom => 0
    ModCapacity => 0

Process finished with exit code 0
```

Як бачимо, за рахунок не великої кількості унікальних елементів відносно початкового розміру таблиці на цьому запуску програми колізій не відбулось

3. Початковий розмір 64. Заповнюється 100 елементів. loadFactor = 1 (хеш таблиця збільшується коли заповнена повністю). S=999. Пару не знайдено:

```
Results for Chaining
    Default => 15
    PseudoRandom => 13
    ModCapacity => 12
Results for OpenAddress
    Default => 23
    PseudoRandom => 18
    ModCapacity => 24

Process finished with exit code 0
```

4. Початковий розмір таблиці: 65536, заповнюється 16384 випадкових елементів діапазону [-2147483648; 2147483647]

```
Results for Chaining
    Default => 1842
    PseudoRandom => 1848
    ModCapacity => 1942
Results for OpenAddress
    Default => 1811
    PseudoRandom => 1816
    ModCapacity => 1923
```

ВИСНОВКИ

У даній роботі ми ознайомились із побудовою хеш-таблиць, вивчили можливі варіанти реалізації даної структури в контексті вирішення колізій. Було написано програму яка містить:

- Абстракцію хеш-таблиці Map

- Реалізацію ланцюжкового алгоритму з використанням зв’язного списку Chaining
- Реалізацію з відкритим адресуванням і лінійним зондуванням OpenAddress (крок = 1)
- Обидві реалізації хеш-таблиць під час створення класу отримують ззовні (з точки входу) об’єкт що вираховує індекс елемента при даному значенні ключа HashProvider.
- Наведено 3 реалізації HashProvider для цілих чисел:
    - з використанням вбудованої в JVM реалізації Object::hashCode - **Default**
    - псевдо-випадкова функцію побудована на операціях AND, XOR і знакового/беззнакового побітового зсуву - **PseudoRandom**
    - значення самого ключа по модулю розміру таблиці **ModCapacity**
- Домовляємося тримати розмір таблиці завжди степінню двійки

За допомогою наведених вище пунктів було розв’язано задачу про пошук двох чисел що дають в сумі S.

Після кожного пробігу алгоритму підраховується кількість колізій.

З результатів можна побачити що при відносно невеликому заповненні таблиці (16384/65536 = 25%) кількість колізій майже однакова у перших двух хеш-функцій (Default, PseudoRandom) і трохи гірша у останньої. З точки зору алгоритму вирішення колізій при низькій завантаженості таблиці обидва алгоритми показують майже однаковий результат. При високій же загруженості хеш таблиці варіант з відкритою адресою має на 35% більше колізій ніж ланцюжковий алгоритм.

Також можна зробити висновок що якщо обсяг оперативної памяті в даному середовищі не є у дефіциті, збільшення розміру масиву (підтримуючи наприклад 40% чи більше комірок вільними) дуже зменшить кількість колізій а отже зробить її роботу більш ефективною.