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 № 4

з дисципліни «Теорія алгоритмів»

на тему «Піраміди»

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

Мета: визначення послідовності медіан для заданого вхідного масиву.
Завдання: В даній роботі необхідно розв'язати наступну задачу визначення послідовності медіан для заданого вхідного масиву. Нагадаємо, що медіаною для масиву називається елемент, який займає середнє положення у відсортованому масиві. Так, якщо кількість елементів у масиві непарна, то медіана одна та індекс її у відсортованому масиві визначається як [n/2] (де n — розмір вхідного масиву). Якщо кількість елементів у масиві парна, то медіан буде дві та їх індекси визначаються за формулами [n/2] та [n/2] + 1. Задача формулюється наступним чином. Нехай заданий вхідний масив A = [x1, ..., xN]. Припустимо, що елементи масиву поступають на вхід програми послідовно: в кожний момент часу розглядається новий елемент xi. Необхідно для кожного i (від 1 до N) визначити медіану підмасиву A' = [x1, ..., xi], тобто медіану для масиву елементів, які були отримані програмою на даний момент часу. Необхідно розв’язати цю задачу з використанням структури даних пірамід і таким чином, щоб кожна медіана визначалась за час O(log(i)).

ПРОГРАМНИЙ КОД

Файл Heap.kt:

```
package ua.santoni7.l4


/**
 * Тип простої піраміди
 */
sealed class HeapType(
    val comparator: Comparator<Int>,
    val defaultValue: Int,
    val name: String
) {
    object Min : HeapType(Comparator.naturalOrder(), Int.MIN_VALUE, "MinHeap")
    object Max : HeapType(Comparator.reverseOrder(), Int.MAX_VALUE, "MaxHeap")
}


/**
 * Клас, що представляє просту піраміду. Приймає як аргумент [HeapType] - тип піраміди
 * Це може бути мінімальна піраміда, у якої кожна вершина менша за її дочірні вершини, або
```

```
 * максимальна, у якої кожна вершина більша за дочірні елементи. Відмінність у
реалізації цих пірамід полягає у компараторі
 * який порівнює числа.
 */
class Heap(private val maxsize: Int, private val heapType: HeapType) {
    private val arr: IntArray
    var size: Int = 0
        private set
    private val comparator: Comparator<Int> get() = heapType.comparator

    init {
        this.size = 0
        arr = IntArray(this.maxsize + 1)
        arr[0] = heapType.defaultValue
    }

    // Індекс батьківської вершини
    private fun parent(pos: Int): Int {
        return pos / 2
    }

    // Індекс лівої дочірньої вершини
    private fun leftChild(pos: Int): Int {
        return 2 * pos
    }

    // Індекс правої дочірньої вершини
    private fun rightChild(pos: Int): Int {
        return 2 * pos + 1
    }

    // Перевірка чи є вершина листом (тобто не має дочірніх вершин)
    private fun isLeaf(pos: Int): Boolean {
        return pos > (size / 2) && pos <= size
    }

    // Перевірка чи існує вершина
```

```
    private fun exists(pos: Int): Boolean = pos <= size


    // Поміняти місцями вершини i та j
    private fun swap(i: Int, j: Int) {
        val tmp = arr[i]
        arr[i] = arr[j]
        arr[j] = tmp
    }


    /**
     * Нормалізація піраміди починаючи з вершини pos і нижче за структурою
     * У цій процедурі виконується перевірка дотримання умов піраміди (чи то мін. піраміда у якої вершина завжди менша за
     * дочірні вершини, чи навпаки).
     *
     * Складність обумовлена максимальною к-стю рекурсивних викликів даної процедури і складає O(log(N)), так як
     * висота бінарної піраміди не перевищує log(N)
     */
    private fun heapify(pos: Int) {
        if (!isLeaf(pos)) {
            if (exists(leftChild(pos)) && comparator.compare(arr[pos], arr[leftChild(pos)]) > 0 ||
                exists(rightChild(pos)) && comparator.compare(arr[pos], arr[rightChild(pos)]) > 0
            ) {
                if (!exists(rightChild(pos)) || comparator.compare(arr[leftChild(pos)], arr[rightChild(pos)]) < 0) {
                    swap(pos, leftChild(pos))
                    heapify(leftChild(pos))
                } else {
                    swap(pos, rightChild(pos))
                    heapify(rightChild(pos))
                }
            }
        }
    }
```

```
    /**
     * Вставка елементу у піраміду. Спочатку вона ставиться на останнє місце,
після чого піднімається вгору допоки
     * не буде виконана умова піраміди
     *
     * Складність обумовлена максимальною к-стю ітерацій циклу і складає
O(log(N)), так як
     * висота бінарної піраміди не перевищує log(N)
     */
    fun insert(element: Int) {
        if (size >= maxsize) {
            return
        }
        arr[++size] = element
        var current = size

        while (comparator.compare(arr[current], arr[parent(current)]) < 0) {
            swap(current, parent(current))
            current = parent(current)
        }
    }

    // Вивід піраміди у консоль (використовується для відлагодження програми)
    fun print() {
        println("========= ${heapType.name} BEGIN ========= ")
        println("Size=$size")
        if (size == 1) println("SINGLE NODE: ${arr[1]}")
        else for (i in 1..size / 2) {
            print(
                " PARENT : " + arr[i]
                        + " LEFT CHILD : " + (if (exists(leftChild(i)))
arr[leftChild(i)] else "NULL")
                        + " RIGHT CHILD :" + (if (exists(rightChild(i)))
arr[rightChild(i)] else "NULL")
            )
            println()
        }
        println("========= ${heapType.name} END ========= ")
```

```
    }

    /**
     * Видаляє і повертає вершину піраміди
     *
     * Складність обумовлена складністю процедури [heapify] і складає O(log(N))
     */
    fun pop(): Int {
        val popped = arr[FRONT]
        arr[FRONT] = arr[size--]
        heapify(FRONT)
        return popped
    }

    /**
     * Повертає поточну вершину не видаляючи її
     */
    fun peek(): Int = arr[FRONT]

    fun asIterable(): Iterable<Int> = arr.copyOfRange(FRONT, size +
1).asIterable()

    companion object {
        private const val FRONT = 1
    }
}
```

Файл MedianHeap.kt:

```
package ua.santoni7.l4

import kotlin.math.abs

/**
 * Структура що дозволяє обчислити медіану в будь який момент за O(1).
 * Всередині містить дві простих піраміди - [minHeap] та [maxHeap]
```

```
 * [maxHeap] - Зберігає усі елементи менші за поточну медіану у максимальну піраміду (тобто корінь структури зберігає найбільше значення)

 * [minHeap] - Зберігає усі елементи більші за поточну медіану у мінімальну піраміду (тобто корінь структури зберігає найменше значення)

 */

class MedianHeap(val capacity: Int = 1000) {

    class Median(val a: Int, val b: Int? = null) {

        override fun toString(): String {

            return if (b != null) "($a; $b)"

            else "($a)"

        }

        // Представити медіану у вигляді десяткового числа. Якщо медіана складається з 2х елементів - повертає їх середнє

        // арифметичше значення

        fun asDouble(): Double = (b?.toDouble()?.plus(a)?.div(2.0)) ?: a.toDouble()

    }


    // Зберігає усі елементи менші за поточну медіану у максимальну піраміду (тобто корінь структури зберігає найбільше значення)

    private val maxHeap = Heap(capacity, HeapType.Max)


    // Зберігає усі елементи більші за поточну медіану у мінімальну піраміду (тобто корінь структури зберігає найменше значення)

    private val minHeap = Heap(capacity, HeapType.Min)


    private val isEmpty: Boolean

        get() = maxHeap.size == 0 && minHeap.size == 0




    /**

     * Вставка числа n у структуру даних. Асимптотична складність зумовлена складністю процедур вставки у просту піраміду

     * [Heap.insert] а також процедури [rebalanceIfNeeded] а тому дорівнює O(log(N)) + O(log(N)) = O(log(N))

     */

    fun insert(n: Int) {

        if (isEmpty) { // Якщо елементів до цього часу не було, додаємо перший елемент у мінімальну піраміду

            minHeap.insert(n)
```

```
        } else {
            // Якщо n менше чи рівне поточній медіані, додаємо до максимальної піраміди
            // В іншому разі - до мінімальної
            if (n.toDouble().compareTo(median().asDouble()) <= 0) {
                maxHeap.insert(n)
            } else {
                minHeap.insert(n)
            }
        }
        // Перевірка на збалансованість структури у випадку якщо розмір мінімальної і максимальної пірамід відрізняються
        // більш ніж на одиницю
        rebalanceIfNeeded()
    }

    /**
     * Перевірка на збалансованість структури:
     * у випадку якщо розмір мінімальної і максимальної пірамід відрізняються більш ніж на одиницю, видаляємо найперший елемент
     * з більшої піраміди і додаємо його до меншої. В результаті розмір пірамід буде відрізнятись не більше ніж на 1
     *
     * Асимптотична складність зумовлена складністю операцій вставки і видалення простої піраміди [Heap.insert] та [Heap.pop]
     * і дорівнює O(log(N))+O(log(N)) = O(log(N))
     */
    private fun rebalanceIfNeeded() {
        if (abs(maxHeap.size - minHeap.size) > 1) {
            if (maxHeap.size > minHeap.size) {
                minHeap.insert(maxHeap.pop())
            } else {
                maxHeap.insert(minHeap.pop())
            }
        }
    }

    /**
```

```
     * Знаходить поточну медіану. Якщо мін. і макс. піраміди однакового розміру, медіана – це пара чисел (вершини обох пірамід)
     * Якщо одна з пірамід має на один елемент більше ніж інша, медіаною буде вершина цієї піраміди
     *
     * Цей метод не містить циклічних операцій тому його складність O(1)
     */
    fun median(): Median {
        return when {
            maxHeap.size == minHeap.size -> Median(maxHeap.peek(), minHeap.peek())
            maxHeap.size > minHeap.size -> Median(maxHeap.peek())
            else -> Median(minHeap.peek())
        }
    }

    /**
     * Представити структуру у вигляді рядка
     */
    override fun toString(): String {
        val sb = StringBuilder()
        sb.append("\n Median for the numbers : ")
        for (i in maxHeap.asIterable()) {
            sb.append(" $i")
        }
        for (i in minHeap.asIterable()) {
            sb.append(" $i")
        }
        sb.append(" is " + median() + "\n")
        return sb.toString()
    }
}
```

Файл Lab4.kt:

```
package ua.santoni7.l4

import java.util.*
```

```
fun main(){
    val sc = Scanner(System.`in`)
    println("Input N:")
    val n = sc.nextInt()
    val medianHeap = MedianHeap(n)
    val list = mutableListOf<Int>()
    for(i in 1..n){
        // Вводимо масив по 1 елементу і додаємо у структуру medianHeap.
        // Виводимо в консоль поточну медіану масиву що уже введено
        println("Input number #$i: ")
        val x = sc.nextInt()
        medianHeap.insert(x)
        list.add(x)
        println("Median for array ${list.joinToString(prefix = "[", postfix =
"]") { it.toString() }} is ${medianHeap.median()}\n")
    }
}
```

РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМИ

Програма виконується покроково, зчитуючи по одному елементу масиву за раз. Після чого на екран виводиться медіана введеного масиву чисел, у вигляді одного чи двух чисел.

```
Input N:
10
Input number #1:
1
Median for [1] is (1)

Input number #2:
5
Median for [1, 5] is (1; 5)

Input number #3:
7
Median for [1, 5, 7] is (5)

Input number #4:
5
Median for [1, 5, 7, 5] is (5; 5)

Input number #5:
4
Median for [1, 5, 7, 5, 4] is (5)

Input number #6:
6
Median for [1, 5, 7, 5, 4, 6] is (5; 5)

Input number #7:
8
Median for [1, 5, 7, 5, 4, 6, 8] is (5)

Input number #8:
26
Median for [1, 5, 7, 5, 4, 6, 8, 26] is (5; 6)

Input number #9:
34
Median for [1, 5, 7, 5, 4, 6, 8, 26, 34] is (6)

Input number #10:
1
Median for [1, 5, 7, 5, 4, 6, 8, 26, 34, 1] is (5; 6)


Process finished with exit code 0
```

ВИСНОВКИ

У даній роботі ми ознайомились із пірамідальною структурою даних у контексті визначення медіани масиву. Було розроблено програму що виконується покроково, зчитуючи по одному елементу масиву за раз. Після чого на екран виводиться медіана введеного масиву чисел, у вигляді одного чи двух чисел.

У коментарях коду доведено асимптотичну складність даного алгоритму яка сладає O(log(N))

Результати успішної роботи тестової програми наведені вище підтверджують правильність обраних рішень, кінцева мета роботи досягнута.