Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 3 з дисципліни
«Проектування алгоритмів»

„ **Проектування структур даних**”

Виконав(ла) *ІП-13 Нещерет В. О.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О. О.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

2 ЗАВДАННЯ

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД з графічним (не консольним) інтерфейсом користувача (дані БД мають зберігатися на ПЗП), з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

№	**Структура даних**
1	Файли з щільним індексом з перебудовою індексної області, бінарний пошук
2	Файли з щільним індексом з областю переповнення, бінарний пошук
3	Файли з не щільним індексом з перебудовою індексної області, бінарний пошук
4	Файли з не щільним індексом з областю переповнення, бінарний пошук
5	АВЛ-дерево

6	Червоно-чорне дерево
7	B-дерево t=10, бінарний пошук
8	B-дерево t=25, бінарний пошук
9	B-дерево t=50, бінарний пошук
10	B-дерево t=100, бінарний пошук
11	Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук
12	Файли з щільним індексом з областю переповнення, однорідний бінарний пошук
13	Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук
14	Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук
15	АВЛ-дерево
16	Червоно-чорне дерево
17	B-дерево t=10, однорідний бінарний пошук
18	B-дерево t=25, однорідний бінарний пошук
19	B-дерево t=50, однорідний бінарний пошук
20	B-дерево t=100, однорідний бінарний пошук
21	Файли з щільним індексом з перебудовою індексної області, метод Шарра
22	Файли з щільним індексом з областю переповнення, метод Шарра
23	Файли з не щільним індексом з перебудовою індексної області, метод Шарра
24	Файли з не щільним індексом з областю переповнення, метод Шарра
25	АВЛ-дерево
26	Червоно-чорне дерево
27	B-дерево t=10, метод Шарра
28	B-дерево t=25, метод Шарра

29	B-дерево t=50, метод Шарра
30	B-дерево t=100, метод Шарра
31	АВЛ-дерево
32	Червоно-чорне дерево
33	B-дерево t=250, бінарний пошук
34	B-дерево t=250, однорідний бінарний пошук
35	B-дерево t=250, метод Шарра

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

Пошук

```
Function search(key): Key or null
    result = binarySearch(key)
    if (result != null)
        return result
    end if
    if (this.isLeaf())
        return null
    end if
    return childByKey(key).search(key)

Function binarySearch(key): Key or null
    len = records.size
    mid = len / 2
    while (len != 0)
        len /= 2
        if (mid >= 0 and (mid >= keys.size or keys[mid] > key))
            mid -= len / 2 + 1
        end if
        else if (mid < 0 or keys[mid] < key)
            mid += len / 2 + 1
        end else if
        else if (kets[mid] == key)
            return keys[mid]
        end else if
    end while
```

```
        return null
Function childByKey(key): Node
        for (i in records.indices)
                if (keys[i] >= key)
                        return children[i]
                end if
        end for
        return children.last()
```

Вставка

```
Function insert(key): Boolean
        if (root.search(key) == null)
                if (root.keys.size < 2 * t - 1)
                        currentNode = root
                end if
                else
                        currentNode = splitNode(root)
                end else
                while (!currentNode.isLeaf())
                        nextNode = currentNode.childByKey(key)
                        if (nextNode.keys.size < 2 * t - 1)
                                currentNode = nextNode
                        end if
                        else
                                currentNode = splitNode(nextNode)
                        end else
                end while
                currentNode.addKey(key)
                return true
```

```
        end if
        else
            return false
        end else
```

Оновлення

```
Class Record:
    Key : Int
    Data : String

Function searchAndUpdate(record): Boolean {
    result = binarySearch(record.key)
    if (result != null)
        delete(result)
        result.data = record.data
        insert(result)
        return true
    end if
    if (this.isLeaf())
        return false
    end if
    return childByKey(record.key).searchAndUpdate(record)
```

Видалення

```
Function delete(record)
    if (binarySearch(record.key) != null)
        if (isLeaf())
            records.remove(record)
        end if
```

```
        else
            leftChild = childByKey(record.key - 1)
            rightChild = childByKey(record.key + 1)
            if (leftChild.records.size >= t)
                predecessor = leftChild.records.last()
                records.remove(record)
                addRecord(predecessor)
                leftChild.delete(predecessor)
            end if
            else if (rightChild.records.size >= t)
                successor = rightChild.records.first()
                records.remove(record)
                addRecord(successor)
                rightChild.delete(successor)
            end else if
            else
                records.remove(record)
                children.remove(rightChild)
                leftChild.addRecord(record)
                leftChild.addAllRecords(rightChild.records)
                leftChild.children.addAll(rightChild.children)
                leftChild.delete(record)
            end else
        end else
    end if
    else
        nextNode = childByKey(record.key)
        if (nextNode.records.size < t)
            nextNodeIndex = children.indexOf(nextNode)
            if (nextNodeIndex - 1 >= 0)
```

```
        leftSibling =children[nextNodeIndex - 1]
    end if
    else
        leftSibling = null
    end else
    if (nextNodeIndex + 1 <= children.lastIndex)
        rightSibling = children[nextNodeIndex + 1]
    end if
    else
        rightSibling = null
    end else
    if (leftSibling != null and leftSibling.records.size >= t)
        nextNode.addRecord(records[nextNodeIndex - 1])
        records.removeAt(nextNodeIndex - 1)
        addRecord(leftSibling.records.last())
        leftSibling.records.removeLast()
    end if
    else if (rightSibling != null
            and rightSibling.records.size >= t)
        nextNode.addRecord(records[nextNodeIndex])
        records.removeAt(nextNodeIndex)
        addRecord(rightSibling.records.first())
        rightSibling.records.removeFirst()
    end else if
    else
        if (leftSibling != null)
            nextNode.addRecord
                (records[nextNodeIndex - 1])
            records.removeAt(nextNodeIndex - 1)
            children.remove(leftSibling)
```

```
                    nextNode.addAllRecords(leftSibling.records)
                    nextNode.children.addAll(0,
                        leftSibling.children)
                end if
                else if (rightSibling != null)
                    nextNode.addRecord
                        (records[nextNodeIndex])
                    records.removeAt(nextNodeIndex)
                    children.remove(rightSibling)
                    nextNode.addAllRecords
                        (rightSibling.records)
                    nextNode.children.addAll
                        (rightSibling.children)
                end else if
            end else
        end if
    nextNode.delete(record)
    end else
```

3.2 Часова складність пошуку

Процедура пошуку складається з рекурсивного заглиблення по вузлах дерева та бінарного пошуку значення всередині вузла. Загальна складність двох функцій дорівнює $O(\log n * (t + \log t)) = O(t * \log n)$, де n – це кількість вузлів у дереві, а t – це параметр дерева.

3.3 Програмна реалізація

3.3.1 Вихідний код

BTree.kt

```
package com.example.lab3

import android.content.Context
import java.io.Serializable
```

```
class BTree(private val t: Int = 50) : Serializable {

    private var root: Node = Node(t)

    fun search(context: Context, key: Int): String {
        val result = root.search(key)
        return if (result != null)
context.getString(R.string.successful_search, result.toString())
        else context.getString(R.string.no_such_key)
    }

    fun insert(context: Context, record: Record): String {
        return if (root.search(record.key) == null) {
            var currentNode = if (root.records.size < 2 * t - 1) root else
splitNode(root)
            while (!currentNode.isLeaf()) {
                val nextNode = currentNode.childByKey(record.key)
                currentNode =
                    if (nextNode.records.size < 2 * t - 1) nextNode else
splitNode(nextNode)
            }
            currentNode.addRecord(record)
            context.getString(R.string.successful_insert)
        } else context.getString(R.string.unsuccessful_insert)
    }

    private fun splitNode(node: Node): Node {
        val midRecord = node.records[t - 1]

        val leftChildren = node.children.take(t).toMutableList()
        val leftRecords = node.records.take(t - 1).toMutableList()
        val leftNode = Node(t, node.parent, leftRecords, leftChildren)
        for (child in leftChildren) child.parent = leftNode

        val rightChildren = node.children.takeLast(t).toMutableList()
        val rightRecords = node.records.takeLast(t - 1).toMutableList()
        val rightNode = Node(t, node.parent, rightRecords, rightChildren)
        for (child in rightChildren) child.parent = rightNode

        return if (node.parent == null) {
            val newRootChildren = mutableListOf(leftNode, rightNode)
            val newRootRecords = mutableListOf(midRecord)
            val newRoot = Node(t, null, newRootRecords, newRootChildren)
            root = newRoot
            leftNode.parent = newRoot
            rightNode.parent = newRoot
            newRoot
        } else {
            val childIndex = node.parent!!.children.indexOf(node)
            node.parent!!.children.remove(node)
            node.parent!!.addRecord(midRecord)
            node.parent!!.children.addAll(childIndex, mutableListOf(leftNode,
rightNode))
            node.parent!!
        }
    }

    fun update(context: Context, record: Record): String {
        return if (root.searchAndUpdate(record))
            context.getString(R.string.successful_update,
record.key.toString())
        else context.getString(R.string.no_such_key)
    }
```

```
    fun delete(context: Context, key: Int): String {
        val record = root.search(key)
        return if (record != null) {
            root.delete(record)
            context.getString(R.string.successful_delete)
        } else context.getString(R.string.no_such_key)

    }
}
```

Node.kt

```
package com.example.lab3

import android.util.Log
import java.io.Serializable

class Node(
    private val t: Int,
    var parent: Node? = null,
    var records: MutableList<Record> = mutableListOf(),
    var children: MutableList<Node> = mutableListOf()
) : Serializable {

    fun isLeaf() = children.isEmpty()

    fun search(key: Int): Record? {
        val result = searchRecursive(key)
        val comparisons = result.second
        Log.d("SearchTest", "Comparisons : $comparisons.")
        return result.first
    }

    private fun searchRecursive(key: Int): Pair<Record?, Int> {
        var comparisons = 0
        val result = binarySearch(key)
        comparisons += result.second
        if (result.first != null) return Pair(result.first, comparisons)
        if (this.isLeaf()) return Pair(null, comparisons)
        val recursiveResult = childByKey(key).searchRecursive(key)
        comparisons += recursiveResult.second
        return Pair(recursiveResult.first, comparisons)
    }

    fun searchAndUpdate(record: Record): Boolean {
        val result = binarySearch(record.key).first?.let { it.data =
record.data }
        if (result != null) return true
        if (this.isLeaf()) return false
        return childByKey(record.key).searchAndUpdate(record)
    }

    private fun binarySearch(key: Int): Pair<Record?, Int> {
        var comparisons = 0
        var len = records.size
        var mid = len / 2
        while (len != 0) {
            comparisons++
            len /= 2
            if (mid >= 0 && (mid >= records.size || records[mid].key > key))
mid -= len / 2 + 1
            else if (mid < 0 || records[mid].key < key) mid += len / 2 + 1
            else if (records[mid].key == key) return Pair(records[mid],
```

```
comparisons)
        }
        return Pair(null, comparisons)
    }

    fun childByKey(key: Int): Node {
        for (i in records.indices) {
            if (records[i].key >= key) {
                return children[i]
            }
        }
        return children.last()
    }

    fun addRecord(record: Record) {
        records.add(record)
        records.sortBy { it.key }
    }

    fun addAllRecords(records: MutableList<Record>) {
        this.records.addAll(records)
        records.sortBy { it.key }
    }

    fun delete(record: Record) {
        if (binarySearch(record.key).first != null) {
            if (isLeaf()) {
                records.remove(record)
            } else {
                val leftChild = childByKey(record.key - 1)
                val rightChild = childByKey(record.key + 1)
                if (leftChild.records.size >= t) {
                    val predecessor = leftChild.records.last()
                    records.remove(record)
                    addRecord(predecessor)

                    leftChild.delete(predecessor)
                } else if (rightChild.records.size >= t) {
                    val successor = rightChild.records.first()
                    records.remove(record)
                    addRecord(successor)

                    rightChild.delete(successor)
                } else {
                    records.remove(record)
                    children.remove(rightChild)
                    leftChild.addRecord(record)
                    leftChild.addAllRecords(rightChild.records)
                    leftChild.children.addAll(rightChild.children)

                    leftChild.delete(record)
                }
            }
        } else {
            val nextNode = childByKey(record.key)
            if (nextNode.records.size < t) {
                val nextNodeIndex = children.indexOf(nextNode)
                val leftSibling =
                    if (nextNodeIndex - 1 >= 0) children[nextNodeIndex - 1]
                    else null
                val rightSibling =
                    if (nextNodeIndex + 1 <= children.lastIndex)
children[nextNodeIndex + 1]
                    else null
```

```
                if (leftSibling != null && leftSibling.records.size >= t) {
                    nextNode.addRecord(records[nextNodeIndex - 1])
                    records.removeAt(nextNodeIndex - 1)
                    addRecord(leftSibling.records.last())
                    leftSibling.records.removeLast()
                } else if (rightSibling != null && rightSibling.records.size
>= t) {
                    nextNode.addRecord(records[nextNodeIndex])
                    records.removeAt(nextNodeIndex)
                    addRecord(rightSibling.records.first())
                    rightSibling.records.removeFirst()
                } else {
                    if (leftSibling != null) {
                        nextNode.addRecord(records[nextNodeIndex - 1])
                        records.removeAt(nextNodeIndex - 1)
                        children.remove(leftSibling)
                        nextNode.addAllRecords(leftSibling.records)
                        nextNode.children.addAll(0, leftSibling.children)
                    } else if (rightSibling != null) {
                        nextNode.addRecord(records[nextNodeIndex])
                        records.removeAt(nextNodeIndex)
                        children.remove(rightSibling)
                        nextNode.addAllRecords(rightSibling.records)
                        nextNode.children.addAll(rightSibling.children)
                    }
                }
            }
            nextNode.delete(record)
        }
    }
}
```

Record.kt

```
package com.example.lab3

import java.io.Serializable

class Record(val key: Int, var data: String) : Serializable {
    override fun toString(): String {
        return "$key : \"$data\""
    }
}
```

MainActivity.kt

```
package com.example.lab3

import android.os.Bundle
import android.view.KeyEvent
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import androidx.constraintlayout.widget.ConstraintSet
import com.example.lab3.databinding.ActivityMainBinding
import java.io.*


class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
```

```
    private val filename = "tree.bin"
    private var tree = BTree()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.radioGroup.setOnCheckedChangeListener { _, _ ->
chooseInputType() }
        binding.button.setOnClickListener { executeAction() }
        binding.buttonDelete.setOnClickListener { deleteTree() }

        binding.editTextKey.setOnKeyListener { _, keyCode, _ ->
handleKeyEvent(keyCode) }
        binding.editTextData.setOnKeyListener { _, keyCode, _ ->
handleKeyEvent(keyCode) }
    }

    override fun onPause() {
        saveTree(tree)
        super.onPause()
    }

    override fun onResume() {
        tree = loadTree()
        super.onResume()
    }

    fun deleteTree() {
        File(filesDir, filename).delete()
        tree = BTree()
        binding.result.text = getString(R.string.result,
getString(R.string.delete_tree))
    }

    private fun executeAction() {
        val key = binding.editTextKey.text.toString().toIntOrNull()
        val data = binding.editTextData.text.toString()

        val result =
            if (key != null) {
                when (binding.radioGroup.checkedRadioButtonId) {
                    R.id.option_insert -> tree.insert(applicationContext,
Record(key, data))
                    R.id.option_update -> tree.update(applicationContext,
Record(key, data))
                    R.id.option_delete -> tree.delete(applicationContext,
key)
                    R.id.option_search -> tree.search(applicationContext,
key)
                    else -> getString(R.string.invalid_action)
                }
            } else getString(R.string.invalid_key)

        binding.result.text = getString(R.string.result, result)
    }

    private fun chooseInputType() {
        val constraintSet = ConstraintSet()
        constraintSet.clone(binding.parentLayout)

        if (binding.radioGroup.checkedRadioButtonId == R.id.option_search
            || binding.radioGroup.checkedRadioButtonId == R.id.option_delete
```

```
        ) {

            binding.editTextData.visibility = View.GONE
            binding.textData.visibility = View.GONE

            constraintSet.connect(
                R.id.text_key, ConstraintSet.END,
                R.id.parent_layout, ConstraintSet.END
            )
        } else {
            binding.editTextData.visibility = View.VISIBLE
            binding.textData.visibility = View.VISIBLE

            constraintSet.connect(
                R.id.text_key, ConstraintSet.END,
                R.id.text_data, ConstraintSet.START
            )
        }
        constraintSet.applyTo(binding.parentLayout)
    }

    private fun handleKeyEvent(keyCode: Int): Boolean {
        if (keyCode == KeyEvent.KEYCODE_ENTER) {
            binding.button.callOnClick()
            return true
        }
        return false
    }

    private fun saveTree(tree: BTree) {
        val fileOut = FileOutputStream(File(filesDir, filename))
        val objectOut = ObjectOutputStream(fileOut)
        objectOut.writeObject(tree)
        objectOut.close()
    }

    private fun loadTree(): BTree {
        return try {
            val fileIn = FileInputStream(File(filesDir, filename))
            val objectIn = ObjectInputStream(fileIn)
            val obj = objectIn.readObject()
            objectIn.close()
            obj as BTree
        } catch (e: Exception) {
            BTree()
        }
    }
}
```

3.3.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.

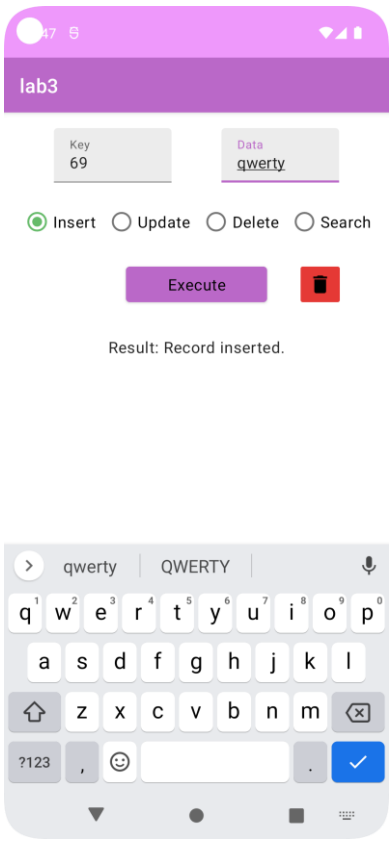

Рисунок 3.1 –Додавання запису

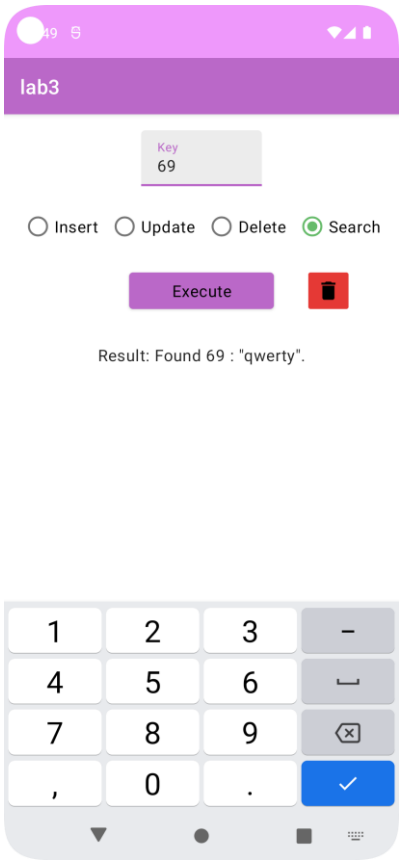

Рисунок 3.2 – Пошук запису

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

Номер спроби пошуку	Число порівнянь
1	14
2	12
3	12
4	15
5	14
6	14
7	12
8	13
9	15
10	13
Середнє число порівнянь : 13,4	

ВИСНОВОК

В рамках лабораторної роботи я реалізував таку структуру даних як B-дерево та використав її для збереження даних у імпровізованій БД. Першим кроком до цього став опис алгоритмів вставки, оновлення, видалення та пошуку значень за допомогою псевдокоду. Далі я виконав програмну реалізацію та додав користувацький інтерфейс. Також я протестував роботу всіх алгоритмів та визначив середню кількість порівнянь при пошуку значення у дереві з 10 000 записів.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 13.11.2022 включно максимальний бал дорівнює – 5. Після 13.11.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 15%;
- аналіз часової складності – 5%;
- програмна реалізація алгоритму – 65%;
- тестування алгоритму – 10%;
- висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.