**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-15, Гуменюк О.В.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.M.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc114359761)

[2 Завдання 4](#_Toc114359762)

[3 Виконання 7](#_Toc114359763)

[3.1 Псевдокод алгоритмів 7](#_Toc114359764)

[3.2 Часова складність пошуку 13](#_Toc114359765)

[3.3 Програмна реалізація 13](#_Toc114359766)

[3.3.1 Вихідний код 13](#_Toc114359767)

[3.3.2 Приклади роботи 21](#_Toc114359768)

[3.4 Тестування алгоритму 24](#_Toc114359769)

[3.4.1 Часові характеристики оцінювання 24](#_Toc114359770)

[Висновок 25](#_Toc114359771)

[Критерії оцінювання 26](#_Toc114359772)

# Мета лабораторної роботи

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

# 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 |

# Виконання

## Псевдокод алгоритмів

**Class** Node:

children: Node[],

keys: Key[],

parent: Node?

**end**

***\*When you add children to parent.children, it is assumed that parent is added to children.parent.***

**Function** search(tree, key): Key or NULL

**if** (tree.isEmpty()) **do**

**return** NULL

**end if**

node = searchNode(tree.root, key)

**if** (node == NULL) **do**

**return** NULL

**end** **if**

**return** binarySearchKey(foundNode.keys, key)

**end**

**Function** searchNode(node, key): Node or NULL

**if** (node.hasKey(key)) **do**

**return** node

**end if**

**else if** (node.isLeaf()) **do**

**return** NULL

**end else if**

**else**

nextNode = node.childByKey(key)

**return** searchNode(nextNode)

**end else**

**end**

**Function** binarySearchKey(keys, key): Key or NULL

high = keys.length

low = 1

**while** (low <= high) **do**

mid = floor(low + (high-low)/2)

**if** (keys[mid] == mid) **do**

**return** keys[mid]

**end if**

**else** **if** (keys[mid] > key) **do**

low = mid + 1

**end else if**

**else**

high = mid - 1

**end else**

**end** **while**

**return** NULL

**end**

**Function** insert(tree, key, t)

**if** (tree.isEmpty) **do**

tree.root = new Node()

tree.root.addKey(key)

**end if**

**else**

node = tree.root

**while** (not node.isLeaf()) **do**

node = node.childByKey(key)

**end while**

node.addKey(key)

restorePropertyInsert(tree, node, t)

**end else**

**end**

**Function** restorePropertyInsert (tree, node, t)

**if** (node.keys.length == 2t – 1) **do**

node1 = new Node()

node1.children = node.getFirstChildren(t)

node1.keys = node.getFirstKeys(t-1)

node2 = new Node()

node2.children = node.getLastChildren(t)

node2.keys = node.getLastKeys(t-1)

midKey = node.keys[t]

**if** (node.isRoot()) **do**

newRoot = new Node()

newRoot.children = { node1, node2}

newRoot.keys = {midKey}

tree.root = newRoot

**end if**

**else**

in node.parent.children r*eplace node with node1 and node2*

node.parent.addKey(midKey)

restorePropertyInsert(tree, node.parent, t)

**end else**

**end if**

**end**

**Function** delete(tree, key, t): Boolean

**if** (tree.isEmpty()) **do**

return false

**end if**

node = searchNode(tree.root, key)

**if** (node == NULL) **do**

**return** false

**end if**

**else if** (node.isLeaf()) **do**

removeLeafKey(node, key)

**end else if**

**else**

leftChild = node.childByKey(key)

**if** (leftChild.keysLength > t-1) **do**

predecessor = findPredecessor(leftChild)

predecessorKey = predecessor.getLastKey()

node.replaceKey(key, predecessorKey)

removeLeafKey(predecessor, predecessorKey, tree)

**end if**

**else**

rightChild = leftChild.rightSibling()

successor = findSuccessor(rightChild)

successorKey = successor.getFirstKey()

node.replaceKey(key, successorKey)

removeLeafKey(successor, successorKey, tree)

**end else**

**end else**

**return** true

**end**

**Function** removeLeafKey(node, key, tree)

node.removeKey(key)

restorePropertyDelete(node, tree)

**end**

**Function** restorePropertyDelete(node, tree)

**if** (node.keysLength < t-1) **do**

**if** (node**.**isRoot()) **do**

**if**(node.keysLength==0 and node.childrenLength > 0) **do**

tree.root = node.children[1]

**end if**

**else if**(node.keysLength == 0) **do**

tree.root = NULL

**end else if**

**end if**

**else if** (not borrowFromLeft(node) and

not borrowFromRight(node)) **do**

mergeNodes(node)

restorePropertyDelete(node.parent)

**end else if**

**end if**

**end**

**Function** borrowFromLeft(node): Boolean

left = node.leftSibling()

**if** (left != NULL and left.keysLength > t – 1) **do**

siblingKey = left.extractLastKey()

siblingChild = left.extractLastChild()

parentKey = node.parent.keyByChild(left)

node.parent.replace(parentKey, siblingKey)

node.keys = parentKey + node.keys

node.children = siblingChild + node.children

**return** true

**end if**

**return** false

**end**

**Function** borrowFromRight(node): Boolean

right = node.rightSibling()

**if** (right!= NULL and right.keysLength > t – 1) **do**

siblingKey = left.extractFirstKey()

siblingChild = left.extractFirstChild()

parentKey = node.parent.keyByChild(right) node.parent.replace(parentKey, siblingKey)

node.keys = node.keys + parentKey

node.children = node.children + siblingChild

**return** true

**end if**

**return** false

**end**

**Function** mergeNodes(node)

left = node.leftSibling()

parent = node.parent

**if** (left != NULL) **do**

parentKey = parent.keyByChild(left)

parent.removeKey(parentKey)

node.children = left.children + node.children

parent.removeChild(left)

node.keys = left.keys + parentKey + node.keys

**end if**

**else do**

right = node.rightSibling()

parentKey = parent.keyByChild(right)

node.children = node.children + right.children

parent.removeChild(right)

node.keys = node.keys + parentKey + right.keys

**end else**

**end**

**Function** findPredecessor(node): Node

**while** (not node.isLeaf()) **do**

node = node.getLastChild()

**end** **while**

**return** node

**end**

**Function** findSuccessor(node): Node

**while** (not node.isLeaf()) **do**

node = node.getFirstChild()

**end** **while**

**return** node

**end**

**Function** editKey(tree, key, newKey, t): Boolean

**if** (tree.isEmpty()) **do**

**return** false

**end if**

**if** (delete(tree, key, t)) **do**

insert(tree, newKey, t)

**return** true

**end if**

**return** false

**end**

## Часова складність пошуку

Процедура пошуку складається з двох основних функцій: searchNode (пошук вузла з даним ключем в дереві) і binarySearchKey (пошук заданого ключа у вузлі). Це можливо, щоб обидві функції мали логарифмічну часову складність )), оскільки перша рухається по дереву пошуку, а друга – це бінарний пошук. Однак у моїй реалізації ще використовується додаткова функція childByKey, яка обирає наступного нащадка для перевірки та використовує часу, хоча, знову ж, це можливо реалізувати цю функцію з логарифмічною часовою складністю. binarySearchKey та childByKey викликається у searchNode, щоб перевірити, чи не знайшли ми потрібний вузол, тому часова складність всієї процедури пошуку дорівнює

, де *n* – це кількість вузлів у дереві, а *t* – це параметр дерева (кількості ключів у вузлі).

## Програмна реалізація

### Вихідний код

package com.example.demo.controller

import com.example.demo.utility.log

import java.lang.Integer.min

class Node(

inputChildren: ArrayList<Node> = ArrayList(0),

inputRecords: ArrayList<Record> = ArrayList(0),

private val tValue: Int): java.io.Serializable{

var parent: Node? = null

private set

private var children = inputChildren

fun getChildren(): ArrayList<Node>{

return children

}

init {

for (child in inputChildren) child.parent = this

parent?.let {parent!!.addChild(this)}

}

private var records = inputRecords

fun getRecords(): ArrayList<Record>{

return records

}

val recordsLength: Int

get(){

return records.size

}

val childrenLength: Int

get(){

return children.size

}

val nodeIndex: Int

get(){

return if (parent == null) -1

else parent!!.children.indexOf(this)

}

val leftSibling: Node?

get(){

if (isRoot() || nodeIndex == 0) return null

return parent!!.children[nodeIndex - 1]

}

val rightSibling: Node?

get(){

if (isRoot() || nodeIndex == parent!!.children.size - 1) return null

return parent!!.children[nodeIndex + 1]

}

val firstRecord: Record

get(){

return records.first()

}

val lastRecord: Record

get(){

return records.last()

}

val firstChild: Node

get(){

return children.first()

}

val lastChild: Node

get(){

return children.last()

}

val maxSubtreeNode: Node

get(){

var max = this

while (!max.isLeaf()) max = max.lastChild

log("found max = $max")

return max

}

val minSubtreeNode: Node

get(){

var min = this

while (!min.isLeaf()) min = min.firstChild

log("found min = $min")

return min

}

fun getChild(index: Int): Node{

return children[index]

}

fun getFirstChildren(number: Int): ArrayList<Node> {

return ArrayList(children.take(number))

}

fun getLastChildren(number: Int): ArrayList<Node> {

return ArrayList(children.takeLast(number))

}

fun getRecord(index: Int): Record{

return records[index]

}

fun getFirstRecords(number: Int): ArrayList<Record> {

return ArrayList(records.take(number))

}

fun getLastRecords(number: Int): ArrayList<Record> {

return ArrayList(records.takeLast(number))

}

fun isAtMax(): Boolean{

return this.records.size == 2\*tValue - 1

}

fun isAtMin(): Boolean{

return this.records.size == tValue - 1

}

fun isLowerThanMin(): Boolean{

return this.records.size < tValue - 1

}

fun isEmpty(): Boolean{

return records.isEmpty()

}

fun isLeaf(): Boolean{

return children.isEmpty()

}

fun isRoot(): Boolean{

return parent == null

}

fun hasKey(key: Int): Boolean{

return binarySearchKey(key) != null

}

fun addAllRecordsEnd(recs: ArrayList<Record>){

records.addAll(recs)

}

fun addAllRecordsStart(recs: ArrayList<Record>){

records.addAll(0, recs)

}

fun addRecord(record: Record){

for (i in records.indices){

if (records[i].key > record.key){

records.add(i, record)

return

}

}

records.add(record)

}

fun addRecordEnd(record: Record){

records.add(record)

}

fun addRecordStart(record: Record){

records.add(0, record)

}

fun addChild(node: Node){

for (i in children.indices){

if (children[i].firstRecord.key > node.firstRecord.key){

children.add(i, node)

return

}

}

children.add(node)

node.parent = this

}

fun addAllChildren(nodes: ArrayList<Node>){

for (node in nodes){

addChild(node)

}

}

fun addAllChildrenEnd(nodes: ArrayList<Node>){

children.addAll(nodes)

for (node in nodes) node.parent = this

}

fun addAllChildrenStart(nodes: ArrayList<Node>){

children.addAll(0, nodes)

for (node in nodes) node.parent = this

}

fun addChildEnd(node: Node){

children.add(node)

node.parent = this

}

fun addChildStart(node: Node){

children.add(0, node)

node.parent = this

}

fun removeChild(node: Node){

children.remove(node)

node.parent = null

}

fun removeRecord(record: Record){

records.remove(record)

}

fun removeRecordByKey(key: Int){

val record = binarySearchKey(key)

record?.let{

records.remove(record)

}

}

fun extractLastRecord(): Record {

return records.removeLast()

}

fun extractFirstRecord(): Record {

return records.removeFirst()

}

fun extractLastChild(): Node{

return children.removeLast()

}

fun extractFirstChild(): Node{

return children.removeFirst()

}

fun replaceRecordByKey(key: Int, replacement: Record){

val record = binarySearchKey(key)

record?.let{

val recordIndex = records.indexOf(it)

records[recordIndex] = replacement

}

}

fun recordByChild(child: Node): Record {

val index = min(records.size - 1, children.indexOf(child))

return records[index]

}

fun childByKey(key: Int): Node{

for (i in records.indices){

if (records[i].key >= key){

return children[i]

}

}

return children.last()

}

fun binarySearchKey(key: Int): Record? {

var low = 0

var high = records.size - 1

while (low <= high) {

val mid = (high - low) / 2 + low

if (records[mid].key > key) {

high = mid - 1

}

else if (records[mid].key == key) {

return records[mid]

}

else {

low = mid + 1

}

}

return null

}

override fun toString(): String{

return "children=${children.map{it.records}}:records=$records:parent=${parent?.records}"

}

fun keysToString(): String{

return java.lang.String.valueOf(records.map{it.key})

}

package com.example.demo.controller

import com.example.demo.utility.log

class Tree(val tValue: Int): java.io.Serializable{

var root: Node? = null

private set

fun isEmpty(): Boolean{

return root == null

}

fun search(key: Int): Record?{

if (isEmpty()) return null

val node = searchNode(root!!, key) ?: return null

return node.binarySearchKey(key)

}

private fun searchNode(node: Node, key: Int): Node?{

return if (node.hasKey(key)) node

else if (node.isLeaf()) null

else{

searchNode(node.childByKey(key), key)

}

}

fun insert(record: Record){

if (isEmpty()){

root = Node(inputRecords=arrayListOf(record), tValue=tValue)

}

else{

var node = root!!

while (!node.isLeaf()) {

node = node.childByKey(record.key)

}

node.addRecord(record)

restorePropertyInsert(node)

}

}

private fun restorePropertyInsert(node: Node){

if (node.isAtMax()){

val node1Children = node.getFirstChildren(tValue)

val node1Records = node.getFirstRecords(tValue - 1)

val node1 = Node(node1Children, node1Records, tValue)

val node2Children = node.getLastChildren(tValue)

val node2Records = node.getLastRecords(tValue - 1)

val node2 = Node(node2Children, node2Records, tValue)

val midRecord = node.getRecord(tValue - 1)

if (node.isRoot()){

val newRootChildren = arrayListOf(node1, node2)

val newRootRecords = arrayListOf(midRecord)

val newRoot = Node(newRootChildren, newRootRecords, tValue=tValue)

root = newRoot

}

else{

val parent = node.parent!!

parent.removeChild(node)

parent.addRecord(midRecord)

parent.addAllChildren(arrayListOf(node1, node2))

restorePropertyInsert(parent)

}

}

}

fun delete(key: Int): Boolean{

if (isEmpty()) return false

val node = searchNode(root!!, key) ?: return false

if (node.isLeaf()) removeLeafKey(node, key)

else {

val leftChild = node.childByKey(key)

if (!leftChild.isAtMin()){

val predecessor = leftChild.maxSubtreeNode

val predecessorRecord = predecessor.lastRecord

node.replaceRecordByKey(key, predecessorRecord)

removeLeafKey(predecessor, predecessorRecord.key)

}

else {

val rightChild = leftChild.rightSibling!!

val successor = rightChild.minSubtreeNode

val successorRecord = successor.firstRecord

node.replaceRecordByKey(key, successorRecord)

removeLeafKey(successor, successorRecord.key)

}

}

return true

}

private fun removeLeafKey(node: Node, key: Int){

node.removeRecordByKey(key)

restorePropertyDelete(node)

}

private fun restorePropertyDelete(node: Node){

if (node.isLowerThanMin()){

if (node.isRoot()){

if (node.isEmpty() && node.childrenLength > 0) {

root = node.firstChild

node.removeChild(root!!)

}

else if(node.isEmpty()){

root = null

}

}

else if(!borrowFromLeft(node) && !borrowFromRight(node)){

mergeNodes(node)

restorePropertyDelete(node.parent!!)

}

}

}

private fun borrowFromLeft(node: Node): Boolean{

val left = node.leftSibling

val parent = node.parent!!

if (left != null && !left.isAtMin()){

log("left has enough records to share!")

val siblingRecord = left.extractLastRecord()

val parentRecord = parent.recordByChild(left)

parent.replaceRecordByKey(parentRecord.key, siblingRecord)

node.addRecordStart(parentRecord)

if (!left.isLeaf()){

val siblingChild = left.extractLastChild()

node.addChildStart(siblingChild)

}

return true

}

return false

}

private fun borrowFromRight(node: Node): Boolean{

val right = node.rightSibling

val parent = node.parent!!

if (right != null && !right.isAtMin()){

log("right has enough records to share!")

val siblingRecord = right.extractFirstRecord()

val parentRecord = parent.recordByChild(right)

parent.replaceRecordByKey(parentRecord.key, siblingRecord)

node.addRecordEnd(parentRecord)

if (!right.isLeaf()){

val siblingChild = right.extractFirstChild()

node.addChildEnd(siblingChild)

}

return true

}

return false

}

private fun mergeNodes(node: Node){

val left = node.leftSibling

val parent = node.parent!!

if (left != null){

val parentRecord = parent.recordByChild(left)

parent.removeRecord(parentRecord)

node.addAllChildrenStart(left.getChildren())

parent.removeChild(left)

node.addRecordStart(parentRecord)

node.addAllRecordsStart(left.getRecords())

}

else{

val right = node.rightSibling!!

log("mergeNodes -- right exists, merging with right=$right")

val parentRecord = parent.recordByChild(node)

parent.removeRecord(parentRecord)

node.addAllChildrenEnd(right.getChildren())

parent.removeChild(right)

node.addRecordEnd(parentRecord)

node.addAllRecordsEnd(right.getRecords())

}

}

fun editRecordData(key: Int, newData: String): Boolean{

if (isEmpty()) return false

val node = searchNode(root!!, key) ?: return false

node.binarySearchKey(key)?.let{

it.data = newData

return true

}

return false

}

fun editRecordKey(key: Int, newKey: Int): Boolean{

if (isEmpty()) return false

val node = searchNode(root!!, key) ?: return false

val data = node.binarySearchKey(key)!!.data

delete(key)

insert(Record(newKey, data))

return true

}

fun getFirstRecords(number: Int): ArrayList<Record>{

val list = ArrayList<Record>()

if (isEmpty()) return list

else{

val queue = arrayListOf(root!!)

var recordsCounter = 0

while (queue.isNotEmpty() && recordsCounter < number){

val current = queue.removeFirst()

list.addAll(current.getRecords())

queue.addAll(current.getChildren())

recordsCounter += current.recordsLength

}

list.sortBy{it.key}

return list

}

}

}

### Приклади роботи

На рисунках 3.1, 3.2, 3.3 показані приклади роботи програми для додавання і пошуку запису.

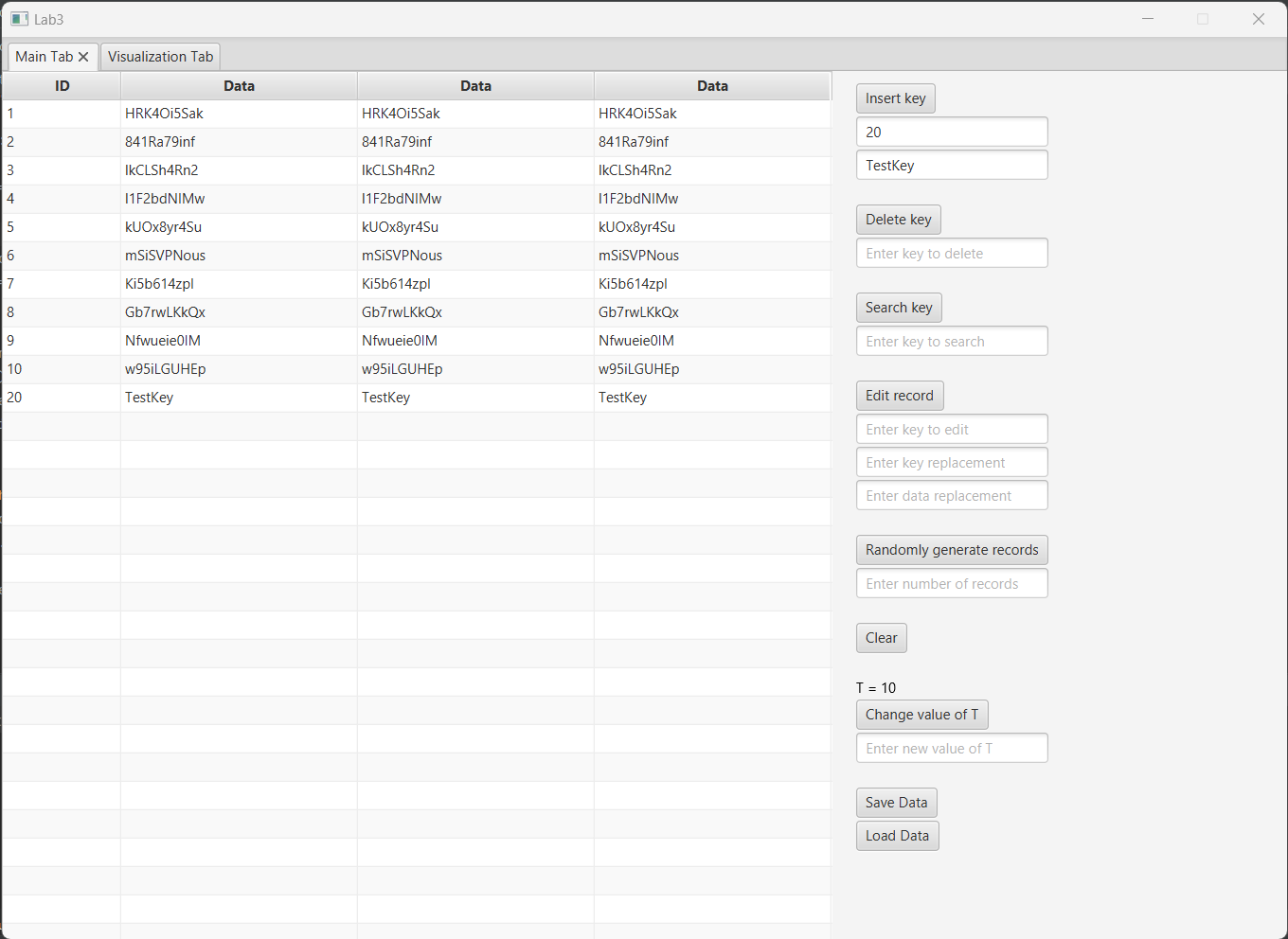


Рисунок 3.1 – Додавання запису

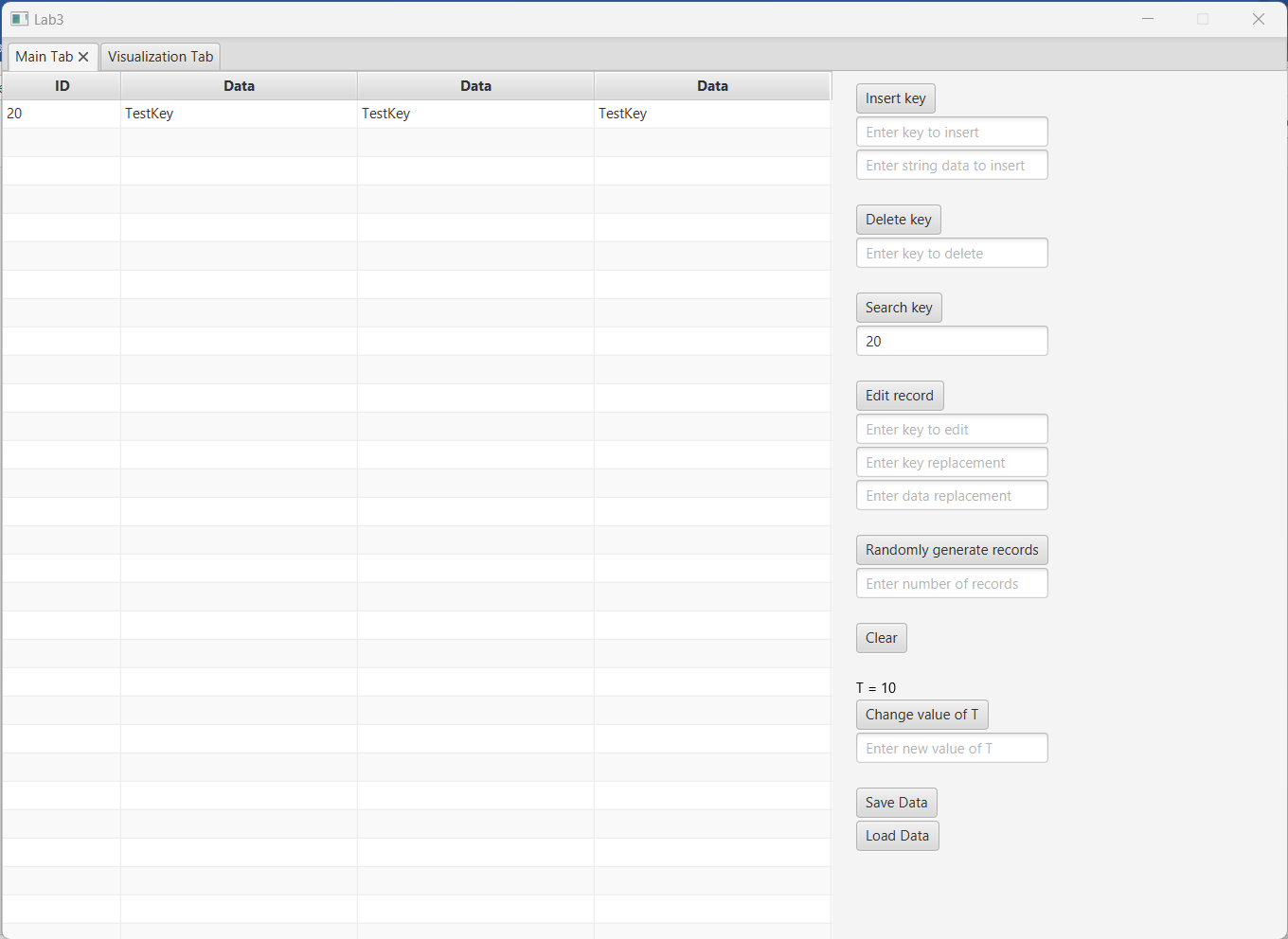


Рисунок 3.2 – Пошук запису

## 

Рисунок 3.3 – Графічне зображення структури дерева

## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу (кількість ключів в дереві = 10 000, t = 10).

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

|  |  |
| --- | --- |
| Номер спроби пошуку | Число порівнянь |
| 1 | 25 |
| 2 | 22 |
| 3 | 24 |
| 4 | 27 |
| 5 | 23 |
| 6 | 34 |
| 7 | 32 |
| 8 | 21 |
| 9 | 26 |
| 10 | 30 |
| 11 | 21 |
| 12 | 27 |
| 13 | 29 |
| 14 | 26 |
| 15 | 33 |
| Середнє значення | 26 |

Висновок

В рамках лабораторної роботи я отримав практичні навички роботи зі складними структурами даних, а саме з B-деревами, та реалізацією простої СУБД. При виконанні даної роботи я записав псевдокод основних алгоритмів, часову складність алгоритму пошуку у дереві, виконав програмну реалізацію власної СУБД з графічним інтерфейсом на основі B-дерев, а також протестував алгоритму пошуку та записав результати тестування.

Псевдокод та програмна реалізація B-дерев були виконані на основі джерел інформації з лекцій курсу і допоміжної літератури. Однак, мій варіант цього алгоритму є дещо спрощеним, оскільки я узагальнив деякі ситуації, які можуть виникнуть при видаленні елементу з дерева. Реалізований алгоритм пошуку ключів у вузлі є класичною та поширеною версією бінарного пошуку. Для проєктування та реалізації графічного інтерфейсу був використаний фреймфорк TornadoFX (на основі JavaFX) та мова програмування Kotlin.

Після написання програми я проаналізував та записав часову складність алгоритму пошуку. У результаті програма виконувала процедуру пошуку з часовою складністю , де *n* – це кількість вузлів у дереві, а *t* – це параметр дерева (кількості ключів у вузлі). Однак, з деякими оптимізаціями допоміжних функцій можливо зменшити часову складність алгоритму до )).

Наприкінці було виконане тестування алгоритму пошуку – у середньому під час роботи алгоритму було виконано 26 порівнянь для *n* = 10000, *t* = 10).

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.