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-24 Ротань Олександр Євгенович

Перевірив
Ахаладзе І. Е.

Київ 2023

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.

Варіант завдання:

19	B-дерево t=50, однорідний бінарний пошук

Виконання

3.1. Псевдокод

```
class BTree {
   Node root
   int t  // Minimum degree

   class Node {
      int n  // Number of keys currently in node
```

```
    object[] key  // An array of keys
    boolean leaf  // Is true when node is leaf. Otherwise false
    Node[] child  // An array of child pointers
}

BTree(t) {
    allocate a new root for BTree
    root.leaf := true
    root.n := 0
    this.t := t
}

function search(Node x, key k) {
    i := 1
    while i <= x.n and k > x.key[i] {
        i := i + 1
    }
    if i <= x.n and k == x.key[i] {
        return (x, i)
    } else if x.leaf {
        return NIL
    } else {
        search(x.child[i], k)
    }
}

function splitChild(Node x, int i, Node y) {
    create new node z
    z.leaf = y.leaf
    z.n = t - 1
```

```
    for j: 1 to t-1 {
        z.key[j] = y.key[j + t]
    }
    if !y.leaf {
        for j: 1 to t {
            z.child[j] = y.child[j + t]
        }
    }
    y.n = t - 1
    for j: x.n + 1 to i + 1 {
        x.child[j + 1] = x.child[j]
    }
    x.child[i + 1] = z
    for j: x.n to i {
        x.key[j + 1] = x.key[j]
    }
    x.key[i] = y.key[t]
    x.n = x.n + 1
}

function insert(key k) {
    Node r = this.root
    if r.n == 2*t - 1 {
        temp := new Node
        this.root = temp
        temp.child[1] = r
        splitChild(temp, 1, r)
        insertNonFull(temp, k)
    } else {
        insertNonFull(r, k)
```

```
    }
}

function insertNonFull(Node x, key k) {
    i = x.n
    if x.leaf {
        while i >= 1 and k < x.key[i] {
            x.key[i + 1] = x.key[i]
            i = i -1
        }
        x.key[i + 1] = k
        x.n = x.n + 1
    } else {
        while i >= 1 and k < x.key[i] {
            i = i - 1
        }
        i = i + 1
        if x.child[i].n == 2*t - 1 {
            splitChild(x, i, x.child[i])
            if k > x.key[i] {
                i = i + 1
            }
        }
        insertNonFull(x.child[i], k)
    }
}

function delete(key k) {
    (Node x, int i) = search(root, k)
    if x == NIL {
```

```
            return
        }
        if x.leaf {
            for j = i to x.n - 1 {
                x.key[j] = x.key[j + 1]
            }
            x.n = x.n - 1
        } else {
            // Complex case handle in this else part
        }
    }
}
function binarySearch(array, targetValue)
    Set start index to 0
    Set end index to the length of the array minus 1
    While start index is less than or equal to end index
        Calculate the middle index
        If target value equals the value in the middle index of array
            Return middle index
        else If target value is less than the value in the middle index of array
            Set end index to middle index minus 1
        else
            Set start index to middle index plus 1
    Return not found
```

3.2. Часова складність пошуку

Часова складність пошуку в B-дереві зазвичай дорівнює $O(\log_m n)$, де n — кількість ключів у дереві, а m.

3.3 Програмна реалізація

3.3.1. Код програми

```
package org.example.lab3.tree;
```

```
import lombok.Data;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.ToString;

import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.MutableTreeNode;
import javax.swing.tree.TreeModel;
import java.awt.*;
import java.io.Serializable;
import java.util.*;

@Data
public class BTree implements Tree, Serializable {

    private final int factor;

    private Node root;

    private final DefaultTreeModel treeModel = new DefaultTreeModel(new DefaultMutableTreeNode("Root"));

    public BTree(int factor) {
        this.factor = factor;
        root = new Node();
    }
```

```
public BTree() {
    this(50);
}

public BTree(Collection<Integer> values) {
    this();
    for (var value : values) {
        insert(value);
    }
}

public BTree(Collection<Integer> values, int factor) {
    this(factor);
    for (var value : values) {
        insert(value);
    }
}

@RequiredArgsConstructor
@Data
private class Node implements Comparable<Node> {

    @ToString.Exclude
    private Node parent;

    private NavigableSet<Integer> values = new TreeSet<>();

    private NavigableSet<Node> children = new TreeSet<>();

    @Override
```

```
public String toString() {
    return "Node{" +
            "parent=" + (parent == null ? "null" : parent.values) +
            ", values=" + values +
            ", children=" + children +
            '}';
}

public Node(int... values) {
    for (var value : values) {
        this.values.add(value);
    }
}

public JTree toJTree() {
    return new JTree(toTreeModel());
}

private DefaultTreeModel toTreeModel() {
    return new DefaultTreeModel(toCell(null));
}

private DefaultMutableTreeNode toCell(MutableTreeNode cellParent) {
    var result = new DefaultMutableTreeNode(values.toString());
    for (var child : children) {
        result.add(child.toCell(result));
    }
    return result;
}
```

```
public boolean isLeaf() {
    return children == null || children.isEmpty();
}

public boolean contains(int key) {
    if (values.contains(key)) return true;
    if (isLeaf()) return false;
    var child = findChildThatContainsKey(key);
    return child.contains(key);
}

private Node split() {
    var left = new Node();
    var right = new Node();
    var middle = values.size() / 2;
    int middleValue;
    var iterator = values.iterator();

    for (int i = 0; i < middle; i++) {
        left.values.add(iterator.next());
    }
    middleValue = iterator.next();
    for (int i = middle + 1; i < values.size(); i++) {
        right.values.add(iterator.next());
    }

    values.clear();

    if (!isLeaf()) {
        var childIterator = children.iterator();
```

```
            for (int i = 0; i <= middle; i++) {
                var child = childIterator.next();
                left.children.add(child);
                child.parent = left;
            }
            for (int i = middle + 1; i < children.size(); i++) {
                var child = childIterator.next();
                right.children.add(child);
                child.parent = right;
            }
            children.clear();
        }

        Node result;
        if (parent == null) {
            result = new Node(middleValue);
            BTree.this.root = result;
        } else {
            result = this.parent.insert(middleValue, false);
            result.children.removeIf((val) -> val.values.isEmpty());
        }

        result.children.add(left);
        result.children.add(right);
        left.parent = result;
        right.parent = result;
        return result;
    }

    public void insert(int key) {
```

```
    insert(key, true);
}

private Node insert(int key, boolean checkChildren) {
    var result = this;
    if (values.size() >= factor * 2 - 1) {
        result = split();
    }
    if (checkChildren) {
        result.insertInner(key);
    } else {
        result.values.add(key);
    }
    return result;
}

private void insertInner(int key) {
    if (isLeaf()) {
        values.add(key);
    } else {
        var iterator = values.iterator();
        int i = 0;
        while (iterator.hasNext() && iterator.next() < key) {
            i++;
        }
        var iteratorChildren = children.iterator();
        for (int j = 0; j < i; j++) {
            iteratorChildren.next();
        }
        var child = iteratorChildren.next();
```

```
            child.insert(key);
        }
    }

    private void merge(@NonNull Node node) {
        values.addAll(node.values);
        children.addAll(node.children);
        node.children.forEach((val) -> val.parent = this);
        if (parent != null) {
            parent.children.remove(node);
        }
    }

    private int minWithRemove() {
        if (isLeaf()) {
            var result = values.first();
            remove(result);
            return result;
        } else {
            return children.first().minWithRemove();
        }
    }

    private int maxWithRemove() {
        if (isLeaf()) {
            var result = values.last();
            remove(result);
            return result;
        } else {
            return children.last().maxWithRemove();
```

```
    }
}


public void remove(int key) {
    if (values.contains(key)) {
        if (isLeaf()) {
            values.remove(key);
        } else {
            var left = findChildThatPrecedesKey(key+1);
            var leftLength = left.values.size();
            if (leftLength > factor - 1) {
                var newKey = left.maxWithRemove();
                values.remove(key);
                values.add(newKey);
            } else {
                var right = findChildThatSucceedsKey(key-1);
                var rightLength = right.values.size();
                if (rightLength > factor - 1) {
                    var newKey = right.minWithRemove();
                    values.remove(key);
                    values.add(newKey);
                } else {
                    left.values.add(key);
                    values.remove(key);
                    left.merge(right);
                    children.remove(right);
                    if (values.isEmpty()) {
                        if (parent == null) {
                            left.parent = null;
```

```
                    BTree.this.root = left;
                } else {
                    parent.children.remove(parent);
                    parent.children.add(this);
                    left.parent = parent;
                }
            }
            left.remove(key);
        }
    }
}
} else {
    var childThatContainsKey = findChildThatContainsKey(key);
    if (childThatContainsKey.values.size() == factor - 1) {
        var donorSibling = findChildThatPrecedesKey(key);
        if (childThatContainsKey != donorSibling &&
donorSibling.values.size() > factor - 1) {
            var precedingKey = findKeyThatPrecedesKey(key);
            var donorKey = donorSibling.values.pollLast();

            childThatContainsKey.values.add(precedingKey);
            values.remove(precedingKey);

            if (!donorSibling.isLeaf()) {

childThatContainsKey.children.add(donorSibling.children.pollLast());
                donorSibling.children.last().parent = childThatContainsKey;
            }

            values.add(donorKey);
```

```
            } else {
                donorSibling = findChildThatSucceedsKey(key);
                var succeedingKey = findKeyThatSucceedsKey(key);
                if (donorSibling.values.size() > factor - 1) {
                    var donorKey = donorSibling.values.pollFirst();

                    childThatContainsKey.values.add(succeedingKey);
                    values.remove(succeedingKey);

                    if (!donorSibling.isLeaf()) {

childThatContainsKey.children.add(donorSibling.children.pollFirst());
                        donorSibling.children.first().parent = childThatContainsKey;
                    }

                    values.add(donorKey);
                } else {
                    childThatContainsKey.values.add(succeedingKey);
                    values.remove(succeedingKey);
                    if (values.isEmpty()) {
                        if (parent == null) {
                            childThatContainsKey.parent = null;
                            BTree.this.root = childThatContainsKey;
                        } else {
                            parent.children.remove(parent);
                            parent.children.add(this);
                            childThatContainsKey.parent = parent;
                        }
                    }
```

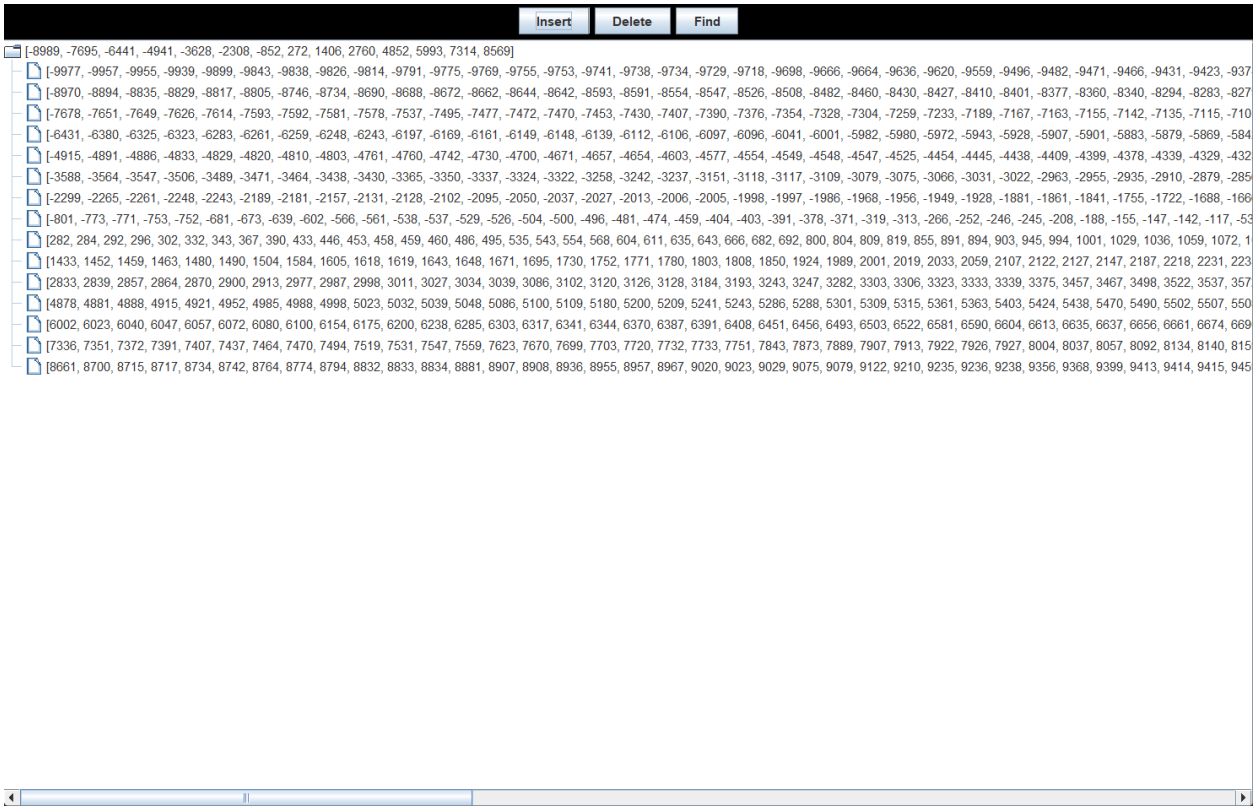
```
                childThatContainsKey.merge(donorSibling);
            }
        }
    }
    childThatContainsKey.remove(key);
  }
}

private Node findChildThatPrecedesKey(int key) {
  return children.lower(new Node(findKeyThatPrecedesKey(key)));
}

private Node findChildThatContainsKey(int key) {
  if (values.first() > key) return children.first();
  if (values.last() < key) return children.last();
  return children.ceiling(new Node(findKeyThatPrecedesKey(key)));
}

private Node findChildThatSucceedsKey(int key) {
  return children.higher(new Node(findKeyThatSucceedsKey(key)));
}

private int findKeyThatPrecedesKey(int key) {
  return Objects.requireNonNullElse(values.lower(key), values.first());
}

private int findKeyThatSucceedsKey(int key) {
  return Objects.requireNonNullElse(values.higher(key), values.last());
}
```

```
    @Override
    public int compareTo(@NonNull Node o) {
        if (o.values == null) return 1;
        if (values == null) return -1;
        if (values.isEmpty()) return o.values.isEmpty() ? 0 : -1;
        if (o.values.isEmpty()) return 1;
        return values.first().compareTo(o.values.first());
    }
}
@Override
public void insert(int key) {
    if (!root.contains(key)) {
        root.insert(key);
    }
}

@Override
public void remove(int key) {
    if (root.contains(key)) {
        root.remove(key);
    }
}

@Override
public boolean contains(int key) {
    return root.contains(key);
}

@Override
public int countStepsToKey(int key) {
```

```
        var result = 0;
        var current = root;
        while (current != null) {
            result++;
            if (current.values.contains(key)) {
                return result;
            }
            current = current.findChildThatContainsKey(key);
        }
        return -1;
    }

    @Override
    public TreeModel toTreeModel() {
        return root.toTreeModel();
    }

    @Override
    public void visualize() {
        var treeVisualization = new TreeVisualization(this);
        treeVisualization.addInsertButton();
        treeVisualization.addRemoveButton();
        treeVisualization.addFindButton();
        treeVisualization.visualize();
    }
}
package org.example.lab3.tree;

import lombok.experimental.UtilityClass;
```

```
@UtilityClass
public class ArraysUtil {

    public static int homogeneousBinarySearch(int[] arr, int key) {
        var delta = arr.length/2;
        var res = arr.length/2;
        var comparsions = 0;
        while (delta > 0 && res >= 0 && res < arr.length) {
            comparsions+=2;
            if (arr[res] == key) {
                System.out.println("Comparsions: " + comparsions);
                return res;
            } else if (arr[res] > key) {
                comparsions++;
                res -= delta;
            } else {
                comparsions++;
                res += delta;
            }
            delta = (delta+1)/2;
            if (res == arr.length) {
                res = arr.length - 1;
            }
        }
        comparsions++;
        System.out.println("Comparsions: " + comparsions);
        return -1;
    }
}
```

3.3.2. Приклад роботи



Зверху знаходиться контрольна панель, за допомогою якої можна видаляти, додавати елементи та перевіряти, чи є елемент в дереві.

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

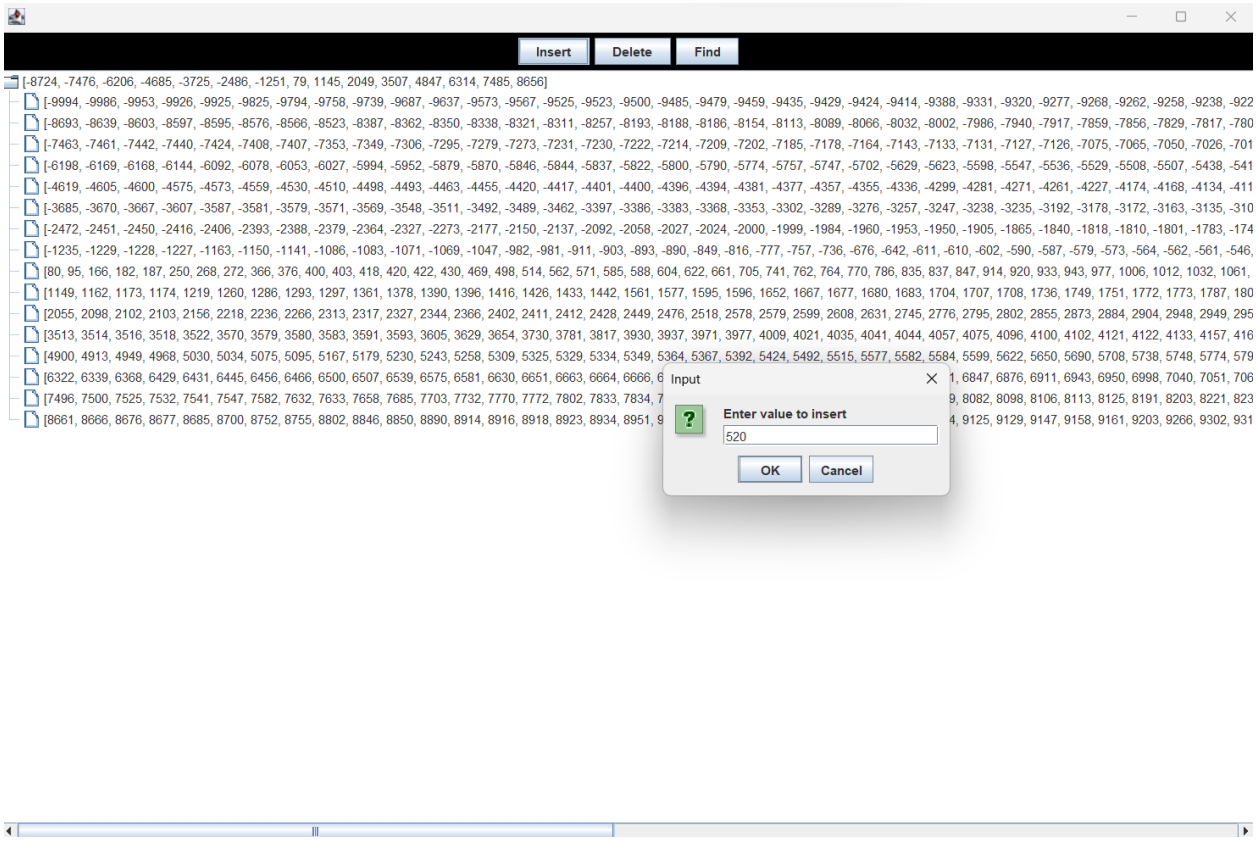


Рисунок 3.1 –Додавання запису

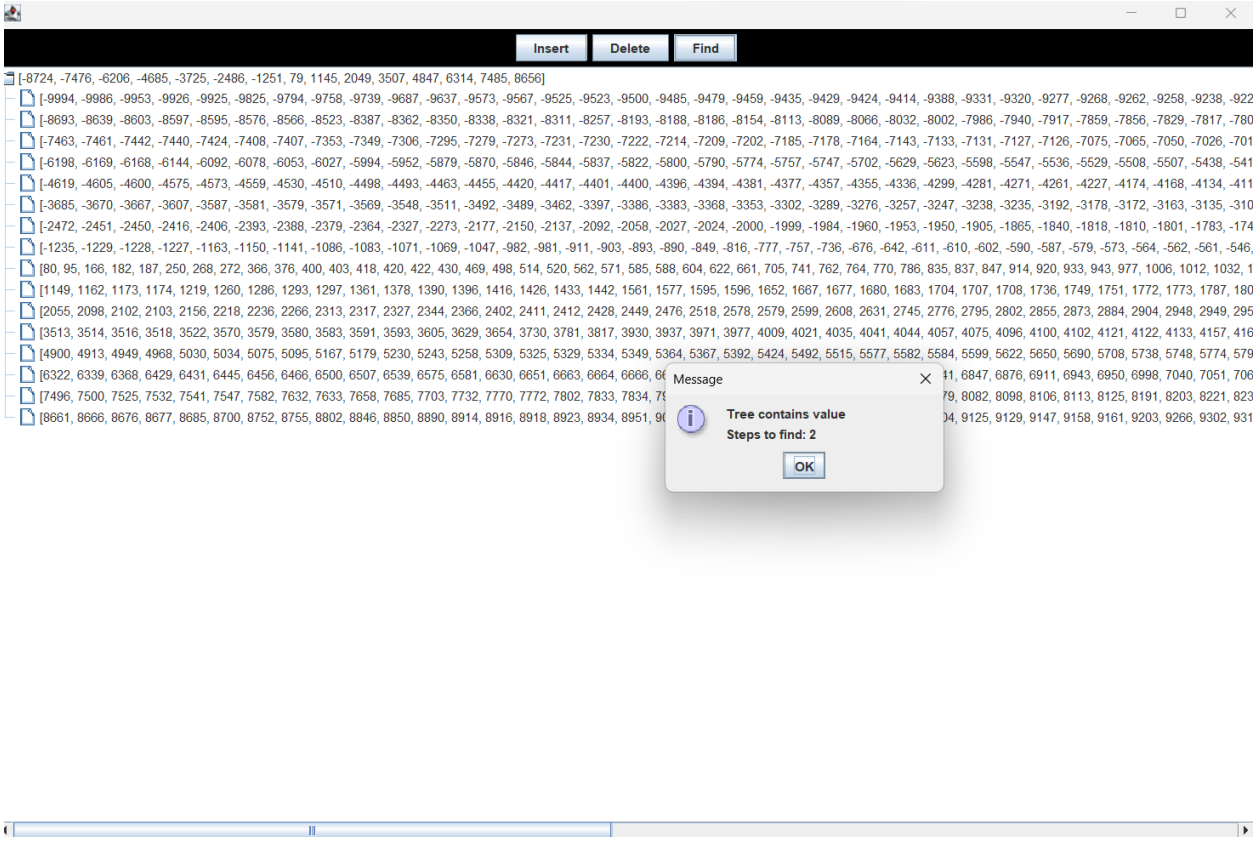


Рисунок 3.2 – Пошук запису

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку елемента в масиві із 100 елементів.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

Номер спроби пошуку	**Число порівнянь**
1	7
2	11
3	17
4	20
5	17
6	7
7	7
8	23
9	14
10	20
11	7
12	7
13	23
14	20
15	23
AVG	14.8(6)

Висновок

Мета цієї лабораторної роботи полягала в ознайомленні з основними підходами проектування та обробки складних структур даних, зокрема з використанням B-дерева з параметром розміру вузла t=50 для вирішення завдань пошуку, додавання, видалення та редагування записів у базі даних. За допомогою псевдокоду були розроблені алгоритми для виконання цих операцій, а також проведена асимптотична оцінка часової складності пошуку в даній структурі даних.

У рамках лабораторної роботи було реалізовано невелику СУБД з графічним інтерфейсом користувача, де дані зберігалися на постійному носії і здійснювалися функції пошуку, додавання, видалення та редагування записів. Було проведено експеримент з заповненням бази даних випадковими значеннями до 10000 записів і зафіксовано середнє число порівнянь для знаходження запису по ключу, яке дозволило оцінити ефективність пошуку у використовуваній структурі даних.