МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
КАФЕДРА КОМПЮТЕРНИХ НАУК

Курсова робота

з дисципліни "Прикладні алгоритми та структури даних" на тему: розробка програмного додатку моделювання та візуалізації подій

Роботу виконав здобувач вищої освіти Вільховик О.О.

Перевірив: _____

Київ, 2020р.

Зміст

Розділ №1 Опис індивідуального завдання

1.1 Загальні технічні вимоги

Розробити та реалізувати програму-симулятор за допомогою якої можна продемонструвати та дослідити основні операції роботи з бінарним деревом. Основними функціональними можливостями є створення двійкового дерева, додавання елемента, виключення елемента та візуалізація обходу дерева використовуючи такі алгоритми: inordered, preordered, postordered.

1.2 Індивідуальна частина

Згідно з індивідуального завдання №8 (код завдання 2211232) програмний додаток повинен обов’язково мати такі функції:

1. Розмір вікна додатку може бути змінений користувачем.
2. Реалізовано алгоритм обходу preordered.
3. Під час анімації обходу дерева всі елементи інтерфейсу залишаються інтерактивними. Якщо будь-яка з операцій визивається під час анімації, тоді вона перериває її та виконується відповідна дія.
4. Вузол якій відвідується під час обходу змінює колір контуру на червоний та товщину контуру до 3.
5. Статичні та невідвідані вузли дерева мають вигляд кола із чорним кольором контуру та шириною 1, текст всередині має чорний колір, вузол не має кольору заливки.
6. Відвідані вузли мають вигляд кола без кольору заливки із сірим кольором контуру та шириною 1, текст всередині має сірий колір.
7. Після завершення анімації обходу всі вузли отримують заливку червоним кольором на 1 секунду, потім оформлення вузла стає стандартним як для статичної візуалізації дерева

Розділ №2 Програмна модель додатку

2.1 Загальна структура проекту

Для реалізації завдання використовується архітектурний шаблон MVC (model view controller) тому весь програмний код розділений на три відповідні пакети. Такий підхід допомагає відділити логіку для відображення користувацького інтерфейсу від внутрішньої логіки виконання програми.

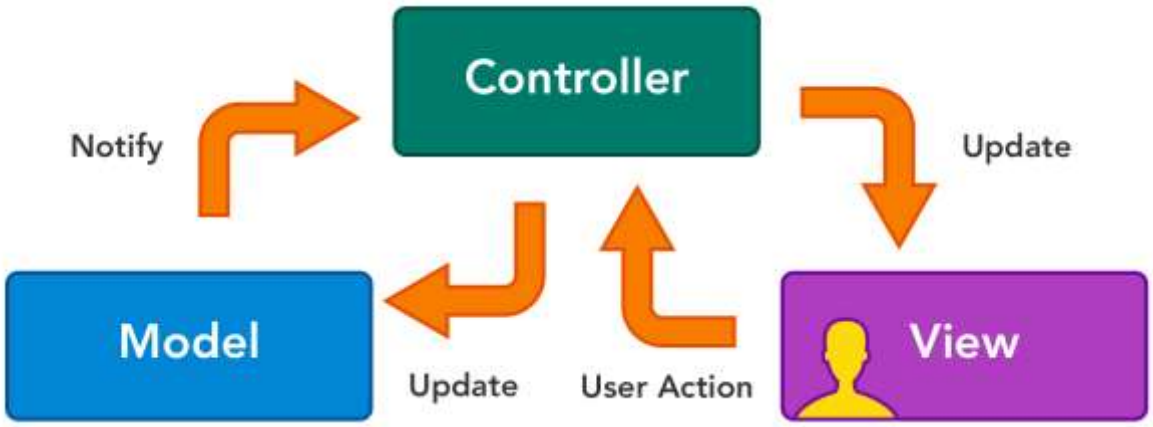


Рисунок 1. Схема архітектурного шаблону MVC та структура проекту

2.2 Опис пакету view

Класи всередині пакету view відповідають за відображення користувацького інтерфейсу та анімації обходу дерева. Інтерфейс складається з головного вікна на якому розташована панель для відображення бінарного дерева (клас BTView). Відображення дерева реалізовано рекурсивною функцією та відбувається від листів до кореня (знизу до гори).

Кожен вузол дерева інкапсульован у клас обгортку NodeViewHolder, методи цього класу дозволяють змінювати відображення будь-якого вузла дерева без перемальовування всього дерева. Використовуючи властивість бінарного дерева, а саме відсутність дублікатів, було створено кеш для вузлів дерева. Кеш являє собою мапу, у якій ключ це значення елемента вузла дерева, а значення – об’єкт класу обгортки NodeViewHolder. Таким чином BTView має доступ до кожного з вузлів дерева та може змінювати їх відображення без перемальовування всього дерева.

Використовуючи кеш об’єктів зручно робити анімацію обходу дерева. Анімація реалізована за допомогою допоміжного потоку тому не блокує інтерфейс користувача під час виконання. Для переривання анімації

використовується метод Thread.interrupt(). Треба зауважити що код який виконується всередині потоку, має механізм для обробки переривань, такий, що у будь-якому стані програма не зможе пошкодити відображення дерева або модель програми.

2.3 Опис пакету model

Класи всередині пакету model реалізують логіку роботи з бінарним деревом пошуку: додавання, видалення, пошук елементів, реалізують алгоритми обходу дерева (inorder, preorder, postorder), та реалізують ітератор для перебирання елементів бінарного дерева. Пакет містить узагальнений інтерфейс, абстрактний клас та їх реалізацію – BST. Така структура моделі дозволяє робити декілька реалізацій бінарних дерев, одною з яких і є бінарне дерево пошуку. Розглянемо основні методи реалізацію класа BST.

2.3.1 Видалення елементу з дерева

Під час видалення з бінарного дерева вузла потрібно розглянути три випадки:

1. елемент знаходиться в лівому піддереві поточного дерева
2. елемент знаходиться в правому піддереві поточного дерева
3. елемент знаходиться в корені поточного дерева

У перших двох випадках потрібно рекурсивно видалити елемент. Якщо елемент знаходиться в корені поточного піддерева і має два дочірніх вузла, то потрібно замінити його мінімальним елементом з правого піддерева і рекурсивно видалити цей мінімальний елемент з правого піддерева. Якщо елемент має один дочірній вузол, потрібно замінити його нащадком.

2.3.2 Пошук елементу у дереві

Для пошуку застосовується головна властивість бінарного дерева, кожен лівий елемент буде меншим ніж даний, а кожний правий – більшим. Таким чином, щоб знайти елементу дереві необхідно порівняти його з поточним елементом у дереві, якщо він більше ніж вхідний – беремо правий елемент для вхідного та продовжуємо порівнювати, якщо поточний елемент менший від вхідного – беремо лівий елемент для вхідного і продовжуємо порівняння, якщо елемент дорівнює поточному – тоді він і є шуканий елемент.

2.3.3 Вставка елемента в дерево

Вставка елемента в дерево виконується аналогічно до пошуку, при виявленні відсутності у елементу лівого або правого піддерева потрібно віконати на його місце вставку.

2.3.4 Операції обходу дерева (inorder, preorder, postorder)

Операція inorder обходу дерева полягає у тому, розташувати всі елементи дерева у порядку зростання. Для цього потрібно спочатку відвідати всі елементи у лівому піддереві, потім кореневий елемент, а потім всі елементи у правому піддереві. Таким чином ми розташуємо всі елементи дерева у порядку їх зростання.

Під час preorder обходу дерева, с початку відвідується кореневий вузол, а потім ліве та праве підерево.

Postorder підхід відвідує спочатку праве та ліве піддерево, а потім кореневий вузол.

3.4 Пакет controller

Пакет містить лише один клас – MainController який виступає посередником між об’єктами view та model. Також контролер приймає на себе користувацькі події, у нашому випадку це кліки по кнопкам та введення тексту, та змушує програму реагувати на них.

Розділ №3 Функціональні можливості

3.1 Графічний інтерфейс додатку

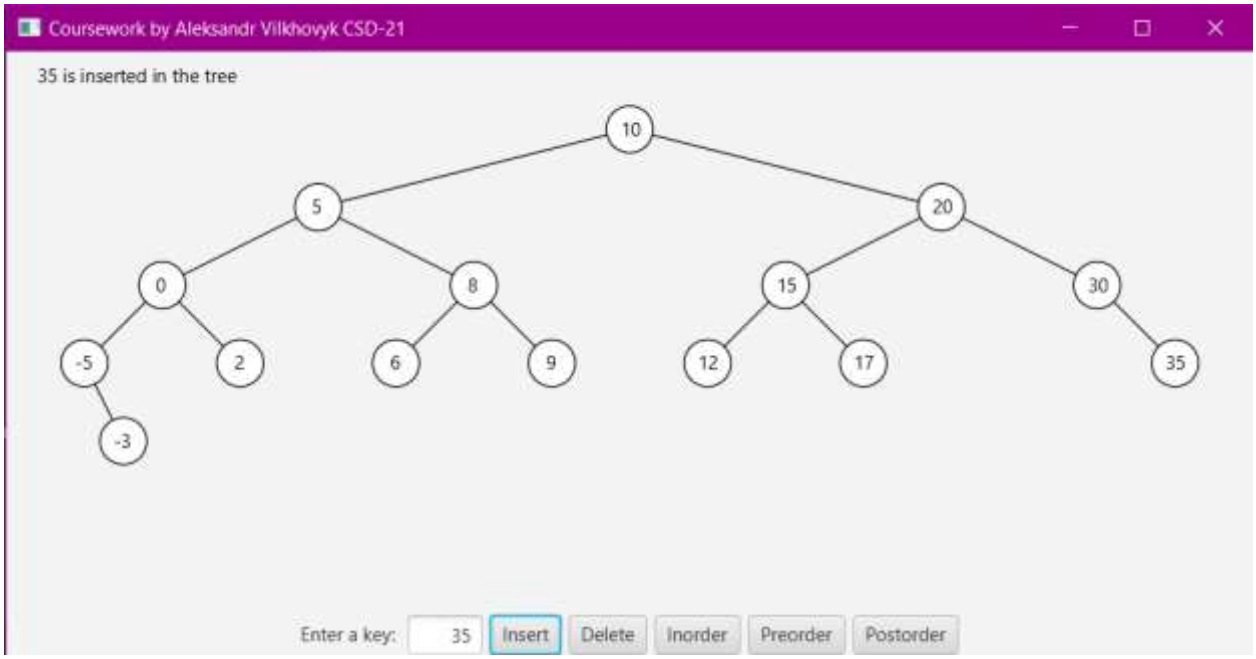


Рисунок 2. Графічний інтерфейс готової програми.

Головне вікно програми має строку стану, відображення бінарного дерева, поле для додавання нового елементу в дерево, кнопки для додавання, видалення та обходу дерева в один із трьох способів.

3.2 Функціональні можливості

Кнопки insert та delete виконують операції додавання та видалення елемента відповідно. Інформація про статус операції записується у строку стану. У разі некоректного вводу програма виведе повідомлення про помилку.

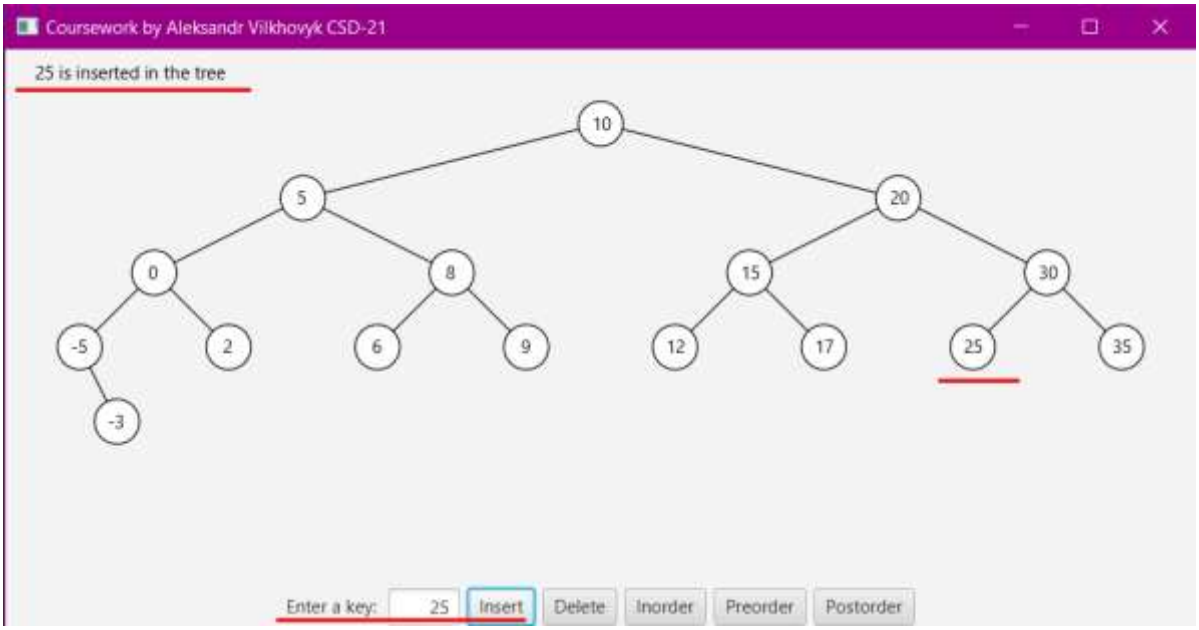


Рисунок 3. Операція вставки елементу.

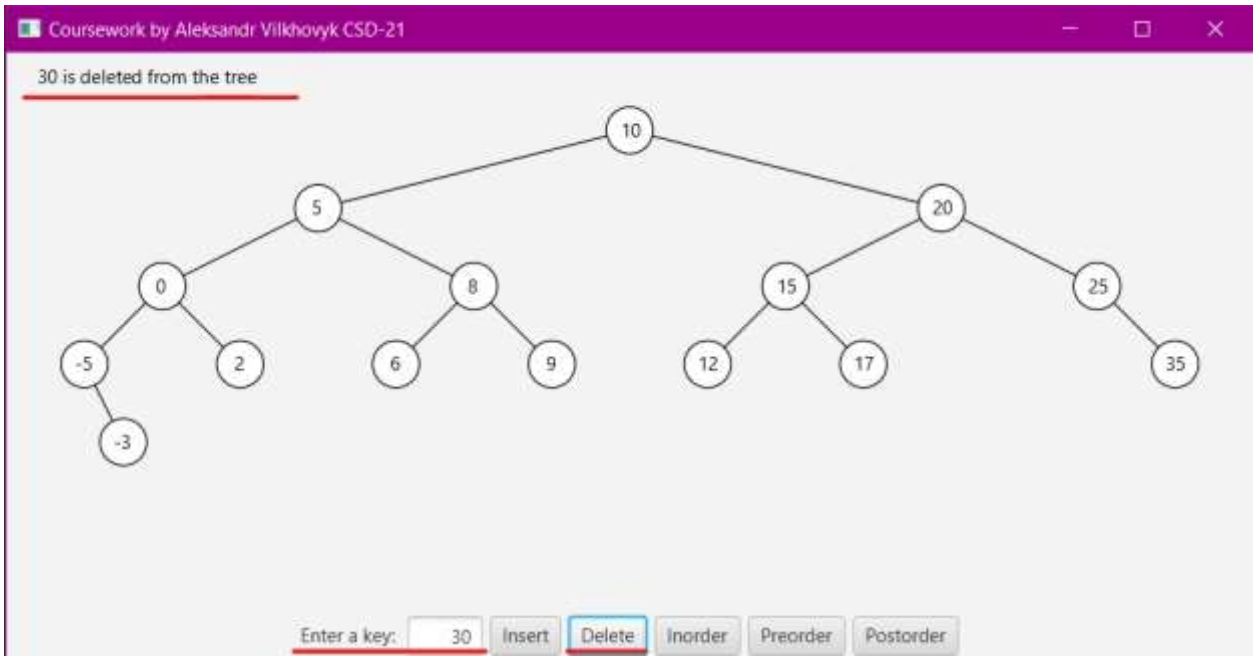


Рисунок 4. Операція видалення елементу.

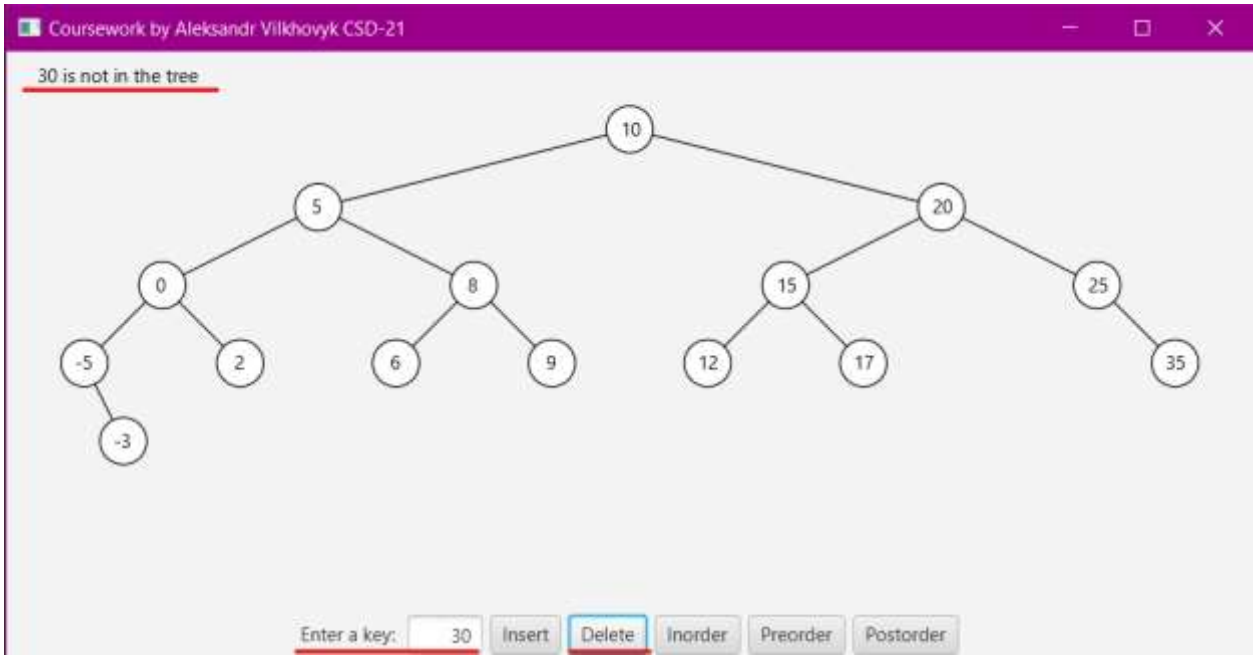


Рисунок 4. Приклад помилки.

Під час виконання операції обходу дерева відповідні вузли підсвічуються червоним контуром відповідно то вимог технічного завдання, а строка стану відображає елементи дерева відсортовані у порядку, відповідному до варіанту обходу дерева.

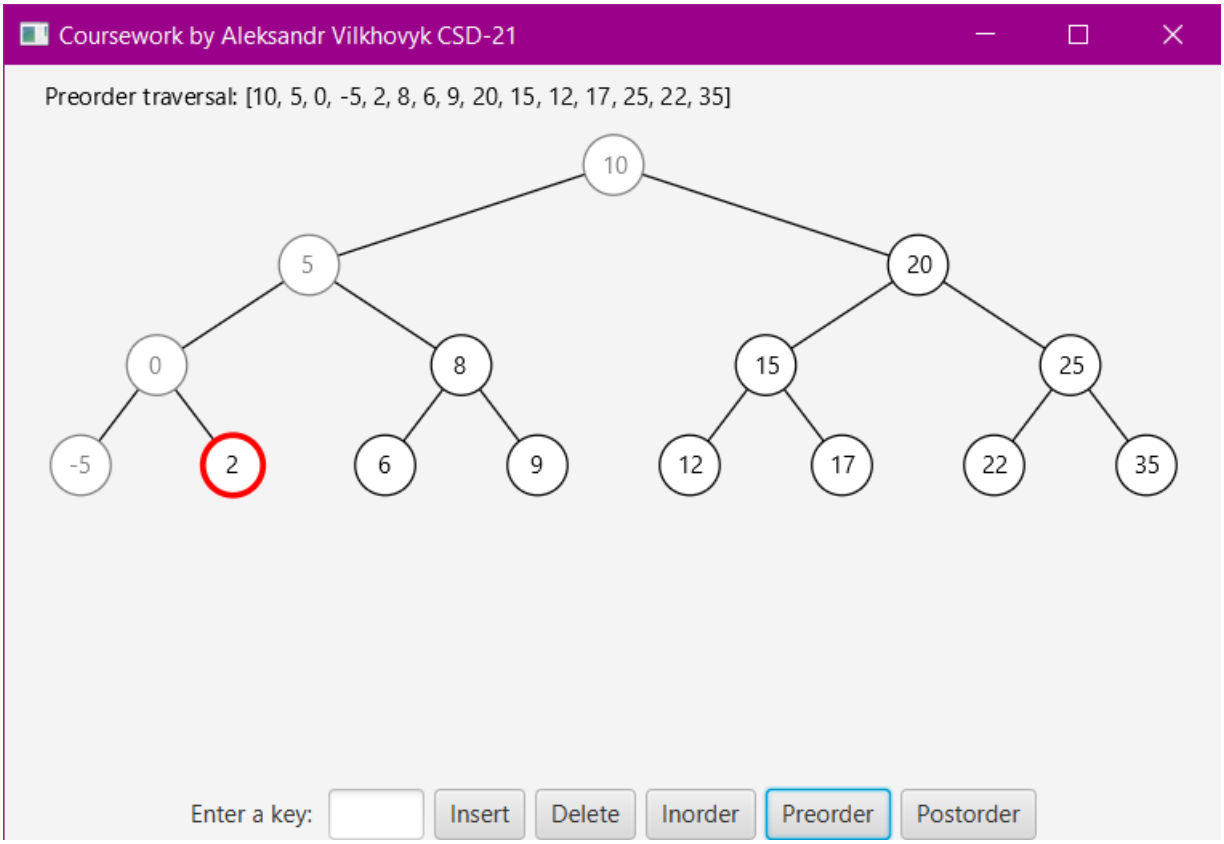


Рисунок 5. Реалізація обходу дерева preorder.

3.3 Висновки

В результаті роботи було розроблено та реалізовано програму-симулятор за допомогою якої можна продемонструвати та дослідити основні операції роботи з бінарним деревом. Були досліджені та реалізовані алгоритми створення двійкового дерева, додавання, виключення елемента та візуалізація обходу дерева за допомогою алгоритмів: inordered, preordered, postordered.

3.4 Перелік літератури

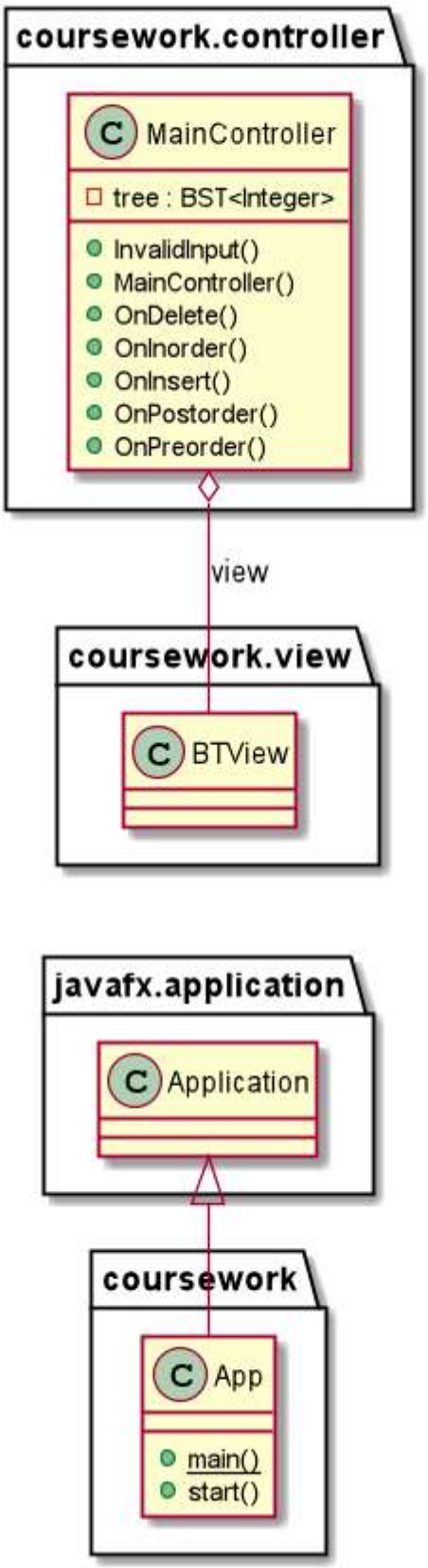
1. Data Structures & Algorithms in Java - Robert Lafore
2. Computer Science: An Interdisciplinary Approach - Robert Sedgewick Kevin Wayne

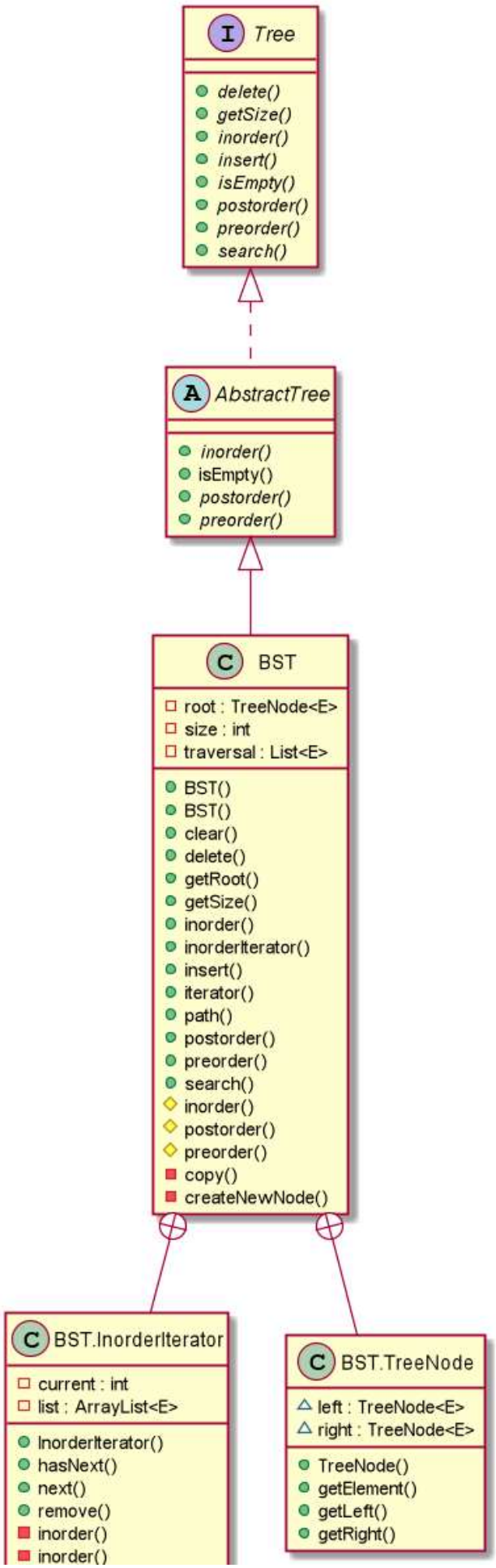
3.5 Посилання

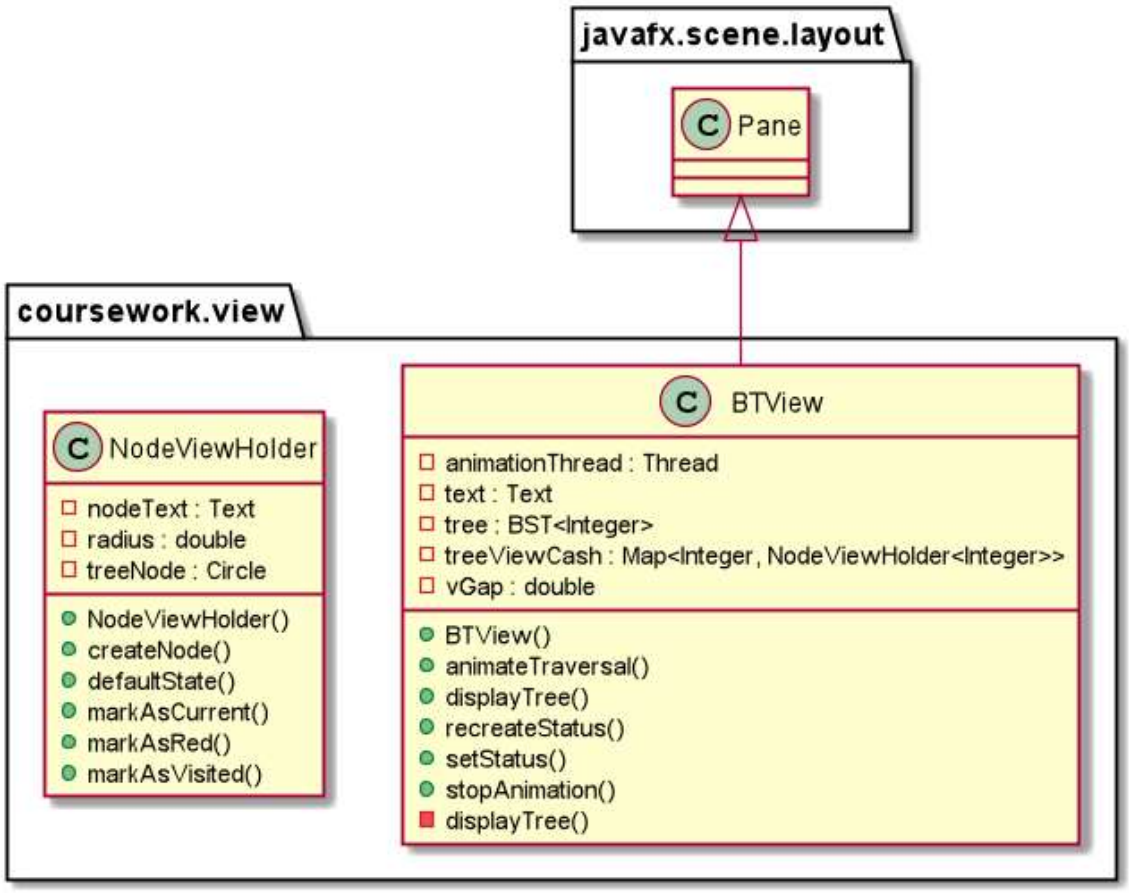
1. Код проекту на GitHub

Розділ №4 Коди класів та відповідні їм UML діаграми

4.1 UML-діаграми класів




Tree
delete()
getSize()
inorder()
insert()
isEmpty()
postorder()
preorder()
search()
AbstractTree
inorder()
isEmpty()
postorder()
preorder()
BST
root : TreeNode<E>
size : int
traversal : List<E>
BST()
BST()
clear()
delete()
getRoot()
getSize()
inorder()
inorderIterator()
insert()
iterator()
path()
postorder()
preorder()
search()
inorder()
postorder()
preorder()
copy()
createNewNode()
BST.InorderIterator
current : int
list : ArrayList<E>
InorderIterator()
hasNext()
next()
remove()
inorder()
inorder()
BST.TreeNode
left : TreeNode<E>
right : TreeNode<E>
TreeNode()
getElement()
getLeft()
getRight()



4.2 Java коди класів программи

```
package coursework;


import coursework.controller.MainController;
import coursework.model.BST;
import coursework.view.BTView;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
```

```
public class App extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override // Override the start method in the Application class
    public void start(Stage primaryStage) {
        BST<Integer> tree = new BST<>(); // Create a tree
        BTView view = new BTView(tree); // Create a View
        MainController controller = new MainController(view, tree);

        BorderPane pane = new BorderPane();
        pane.setCenter(view);

        TextField tfKey = new TextField();
        tfKey.setPrefColumnCount(3);
        tfKey.setAlignment(Pos.BASELINE_RIGHT);
        Button insert = new Button("Insert");
        Button delete = new Button("Delete");
        Button inorder = new Button("Inorder");
        Button preorder = new Button("Preorder");
        Button postorder = new Button("Postorder");
        HBox hBox = new HBox(5);
        hBox.getChildren().addAll(new Label("Enter a key: "),
                tfKey, insert, delete, inorder, preorder, postorder);
        hBox.setAlignment(Pos.CENTER);
        pane.setBottom(hBox);

        insert.setOnAction(e -> {
            try {
                int key = Integer.parseInt(tfKey.getText());
                controller.OnInsert(key);
            } catch (NumberFormatException exception) {
                controller.InvalidInput("Invalid input " + exception);
            }
```

```
        });

        delete.setOnAction(e -> {
            try {
                int key = Integer.parseInt(tfKey.getText());
                controller.OnDelete(key);
            } catch (NumberFormatException exception) {
                controller.InvalidInput("Invalid input " + exception);
            }
        });
        inorder.setOnAction(e -> controller.OnInorder());
        preorder.setOnAction(e -> controller.OnPreorder());
        postorder.setOnAction(e -> controller.OnPostorder());

        // Create a scene and place the pane in the stage
        Scene scene = new Scene(pane, 800, 600);
        primaryStage.setTitle("Coursework by Aleksandr Vilkhovyk CSD-21");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

package coursework.view;

import javafx.collections.ObservableList;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;

public class NodeViewHolder<E> {

    private Circle treeNode = new Circle();
    private Text nodeText = new Text();
    private double radius = 15; // Tree node radius
```

```
    public NodeViewHolder() {}

    public void createNode(ObservableList<Node> root, double x, double y, E
nodeElement) {
        // Display a node
        treeNode = new Circle(x, y, radius);

        // set default state to tree node
        defaultState();

        nodeText = new Text(x - 2 - (nodeElement.toString().length()*2), y +
4, nodeElement.toString());
        root.addAll(treeNode, nodeText);
    }

    public void markAsRed() {
        treeNode.setFill(Color.RED);
    }

    public void markAsCurrent() {
        treeNode.setStrokeWidth(3f);
        treeNode.setStroke(Color.RED);
    }

    public void markAsVisited() {
        treeNode.setStrokeWidth(1f);
        treeNode.setStroke(Color.GRAY);
        nodeText.setFill(Color.GRAY);
    }

    public void defaultState() {
        treeNode.setStroke(Color.BLACK);
        treeNode.setFill(Color.WHITE);
        treeNode.setStrokeWidth(1f);
        nodeText.setFill(Color.BLACK);
    }
}
```

```
package coursework.view;

import coursework.model.BST;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class BTView extends Pane {
    private BST<Integer> tree = new BST<>();
    private Text text;
    // Keeps all instances of displayed tree nodes
    private Map<Integer, NodeViewHolder<Integer>> treeViewCash = new
HashMap<>(8);
    private Thread animationThread;
    private double vGap = 50; // Gap between two levels in a tree

    public BTView(BST<Integer> tree) {
        this.tree = tree;
        recreateStatus("Tree is empty");
    }

    public void recreateStatus(String msg) {
        text = new Text(20, 20, msg);
        getChildren().add(text);
    }

    public void setStatus(String error) {
        text.setText(error);
    }

    public void displayTree() {
```

```
        this.getChildren().clear(); // Clear the pane
        treeViewCash.clear();
        if (tree.getRoot() != null) {
            // Display tree recursively
            displayTree(tree.getRoot(), getWidth() / 2, vGap,
                    getWidth() / 4);
        }
    }

    /** Display a subtree rooted at position (x, y) */
    private void displayTree(BST.TreeNode<Integer> root, double x, double y,
double hGap) {
        if (root.getLeft() != null) {
            // Draw a line to the left node
            getChildren().add(new Line(x - hGap, y + vGap, x, y));
            // Draw the left subtree recursively
            displayTree(root.getLeft(), x - hGap, y + vGap, hGap / 2);
        }

        if (root.getRight() != null) {
            // Draw a line to the right node
            getChildren().add(new Line(x + hGap, y + vGap, x, y));
            // Draw the right subtree recursively
            displayTree(root.getRight(), x + hGap, y + vGap, hGap / 2);
        }

        // create and init view holder for tree node
        NodeViewHolder<Integer> viewHolder = new NodeViewHolder<>();
        viewHolder.createNode(getChildren(), x, y, root.getElement());
        // put view holder to view cash using element value as a kay
        treeViewCash.put(root.getElement(), viewHolder);
    }

    public void animateTraversal(List<Integer> animatedItems) {
        stopAnimation();
        animationThread = new Thread(() -> {
```

```
            try {
                for(Integer key : animatedItems) {
                    // highlight current item
                    treeViewCash.get(key).markAsCurrent();
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    treeViewCash.get(key).markAsVisited();
                }

                treeViewCash.forEach((key, viewHolder) -> {
                    viewHolder.markAsRed();
                });
                Thread.sleep(1000);
                treeViewCash.forEach((key, viewHolder) -> {
                    viewHolder.defaultState();
                });
            } catch (Exception e) {
                treeViewCash.forEach((key, viewHolder) -> {
                    viewHolder.defaultState();
                });
            }
        });
        animationThread.start();
    }

    public void stopAnimation() {
        if(animationThread != null) {
            animationThread.interrupt();
        }
    }
}

package coursework.model;
```

```
import java.util.List;

public interface Tree<E extends Comparable<E>> extends Iterable<E> {
  /** Return true if the element is in the tree */
  public boolean search(E e);

  /** Insert element o into the binary tree
   * Return true if the element is inserted successfully */
  public boolean insert(E e);

  /** Delete the specified element from the tree
   * Return true if the element is deleted successfully */
  public boolean delete(E e);

  /** Inorder traversal from the root*/
  public List<E> inorder();

  /** Postorder traversal from the root */
  public List<E> postorder();

  /** Preorder traversal from the root */
  public List<E> preorder();

  /** Get the number of nodes in the tree */
  public int getSize();

  /** Return true if the tree is empty */
  public boolean isEmpty();
}
```

```
package coursework.model;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```

```
public class BST<E extends Comparable<E>> extends AbstractTree<E> {
    private TreeNode<E> root;
    private List<E> traversal = new ArrayList<>(8);
    private int size = 0;

    /** Create a default binary tree */
    public BST() { }

    /** Create a binary tree from an array of objects */
    public BST(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            insert(objects[i]);
    }



    /** Returns true if the element is in the tree */
    public boolean search(E e) {
        TreeNode<E> current = root; // Start from the root

        while (current != null) {
            if (e.compareTo(current.element) < 0) {
                current = current.left;
            }
            else if (e.compareTo(current.element) > 0) {
                current = current.right;
            }
            else // element matches current.element
                return true; // Element is found
        }

        return false;
    }

    /** Insert element o into the binary tree
```

```
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
    if (root == null)
        root = createNewNode(e); // Create a new root
    else {
        // Locate the parent node
        TreeNode<E> parent = null;
        TreeNode<E> current = root;
        while (current != null)
            if (e.compareTo(current.element) < 0) {
                parent = current;
                current = current.left;
            }
            else if (e.compareTo(current.element) > 0) {
                parent = current;
                current = current.right;
            }
            else
                return false; // Duplicate node not inserted

        // Create the new node and attach it to the parent node
        if (e.compareTo(parent.element) < 0)
            parent.left = createNewNode(e);
        else
            parent.right = createNewNode(e);
    }

    size++;
    return true; // Element inserted
}

private TreeNode<E> createNewNode(E e) {
    return new TreeNode<E>(e);
}

/** Inorder traversal from the root*/
```

```
public List<E> inorder() {
    if(!traversal.isEmpty()) {
        traversal.clear();
    }
    inorder(root);
    return traversal;
}

/** Inorder traversal from a subtree */
protected void inorder(TreeNode<E> root) {
    if (root == null) return;
    inorder(root.left);
    // add element to result list
    traversal.add(root.element);
    inorder(root.right);
}

/** Postorder traversal from the root */
public List<E> postorder() {
    if(!traversal.isEmpty()) {
        traversal.clear();
    }
    postorder(root);
    return traversal;
}

/** Postorder traversal from a subtree */
protected void postorder(TreeNode<E> root) {
    if (root == null) return;
    postorder(root.left);
    postorder(root.right);
    traversal.add(root.element);
}

/** Preorder traversal from the root */
public List<E> preorder() {
```

```
    if(!traversal.isEmpty()) {
        traversal.clear();
    }
    preorder(root);
    return traversal;
}

/** Preorder traversal from a subtree */
protected void preorder(TreeNode<E> root) {
    if (root == null) return;
    traversal.add(root.element);
    preorder(root.left);
    preorder(root.right);
}

/** Inner class tree node */
public static class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E e) {
        element = e;
    }

    public TreeNode<E> getLeft() {
        return left;
    }

    public TreeNode<E> getRight() {
        return right;
    }

    public E getElement() {
        return element;
    }
```

```
}

/** Get the number of nodes in the tree */
public int getSize() {
    return size;
}

/** Returns the root of the tree */
public TreeNode<E> getRoot() {
    return root;
}

/** Returns a path from the root leading to the specified element */
public java.util.ArrayList<TreeNode<E>> path(E e) {
    java.util.ArrayList<TreeNode<E>> list =
            new java.util.ArrayList<TreeNode<E>>();
    TreeNode<E> current = root; // Start from the root

    while (current != null) {
        list.add(current); // Add the node to the list
        if (e.compareTo(current.element) < 0) {
            current = current.left;
        }
        else if (e.compareTo(current.element) > 0) {
            current = current.right;
        }
        else
            break;
    }

    return list; // Return an array of nodes
}

/** Delete an element from the binary tree.
 * Return true if the element is deleted successfully
 * Return false if the element is not in the tree */
```

```
public boolean delete(E e) {
    // Locate the node to be deleted and also locate its parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null) {
        if (e.compareTo(current.element) < 0) {
            parent = current;
            current = current.left;
        }
        else if (e.compareTo(current.element) > 0) {
            parent = current;
            current = current.right;
        }
        else
            break; // Element is in the tree pointed by current
    }

    if (current == null)
        return false; // Element is not in the tree

    // Case 1: current has no left children
    if (current.left == null) {
        // Connect the parent with the right child of the current node
        if (parent == null) {
            root = current.right;
        }
        else {
            if (e.compareTo(parent.element) < 0)
                parent.left = current.right;
            else
                parent.right = current.right;
        }
    }
    else {
        // Case 2: The current node has a left child
        // Locate the rightmost node in the left subtree of
```

```
            // the current node and also its parent
            TreeNode<E> parentOfRightMost = current;
            TreeNode<E> rightMost = current.left;

            while (rightMost.right != null) {
                parentOfRightMost = rightMost;
                rightMost = rightMost.right; // Keep going to the right
            }

            // Replace the element in current by the element in rightMost
            current.element = rightMost.element;

            // Eliminate rightmost node
            if (parentOfRightMost.right == rightMost)
                parentOfRightMost.right = rightMost.left;
            else
                // Special case: parentOfRightMost == current
                parentOfRightMost.left = rightMost.left;
        }

        size--;
        return true; // Element inserted
    }

    /** Obtain an iterator. Use inorder. */
    public Iterator<E> iterator() {
        return inorderIterator();
    }

    /** Obtain an inorder iterator */
    public Iterator<E> inorderIterator() {
        return new InorderIterator();
    }

    // Inner class InorderIterator
    class InorderIterator implements Iterator<E> {
```

```
// Store the elements in a list
private ArrayList<E> list = new ArrayList<E>();
private int current = 0; // Point to the current element in list

public InorderIterator() {
    inorder(); // Traverse binary tree and store elements in list
}

/** Inorder traversal from the root*/
private void inorder() {
    inorder(root);
}

/** Inorder traversal from a subtree */
private void inorder(TreeNode<E> root) {
    if (root == null)return;
    inorder(root.left);
    list.add(root.element);
    inorder(root.right);
}

/** Next element for traversing? */
public boolean hasNext() {
    if (current < list.size())
        return true;

    return false;
}

/** Get the current element and move cursor to the next */
public E next() {
    return list.get(current++);
}

/** Remove the current element and refresh the list */
public void remove() {
```

```
            delete(list.get(current)); // Delete the current element
            list.clear(); // Clear the list
            inorder(); // Rebuild the list
        }
    }

    /** Remove all elements from the tree */
    public void clear() {
        root = null;
        size = 0;
    }

    private void copy(TreeNode<E> root, BST<E> tree) {
        if (root != null) {
            tree.insert(root.element);
            copy(root.left, tree);
            copy(root.right, tree);
        }
    }
}

package coursework.model;

import java.util.List;

public abstract class AbstractTree<E extends Comparable<E>> implements
Tree<E> {

    @Override /** Inorder traversal from the root*/
    public abstract List<E> inorder();

    @Override /** Postorder traversal from the root */
    public abstract List<E> postorder();

    @Override /** Preorder traversal from the root */
    public abstract List<E> preorder();
```

```
    @Override /** Return true if the tree is empty */
    public boolean isEmpty() {
        return getSize() == 0;
    }
//
//  @Override /** Return an iterator for the tree */
//  public java.util.Iterator<E> iterator() {
//    return null;
//  }
}
```

```
package coursework.controller;

import coursework.model.BST;
import coursework.view.BTView;

import java.util.List;

public class MainController {

    private BTView view;
    private BST<Integer> tree;

    public MainController(BTView view, BST<Integer> tree) {
        this.view = view;
        this.tree = tree;
    }

    public void OnInsert(int key) {
        view.stopAnimation();
        if (tree.search(key)) { // key is in the tree already
            view.displayTree();
            view.recreateStatus(key + " is already in the tree");
        } else {
            tree.insert(key); // Insert a new key
```

```
            view.displayTree();
            view.recreateStatus(key + " is inserted in the tree");
        }
    }

    public void OnDelete(int key) {
        view.stopAnimation();
        if (!tree.search(key)) { // key is not in the tree
            view.displayTree();
            view.recreateStatus(key + " is not in the tree");
        } else {
            tree.delete(key); // Delete a key
            view.displayTree();
            view.recreateStatus(key + " is deleted from the tree");
        }
    }

    public void OnInorder() {
        view.stopAnimation();
        List<Integer> traversal = tree.inorder();
        view.setStatus("Inorder traversal: " + traversal.toString());
        view.animateTraversal(traversal);
    }

    public void OnPreorder() {
        view.stopAnimation();
        List<Integer> traversal = tree.preorder();
        view.setStatus("Preorder traversal: " + traversal.toString());
        view.animateTraversal(traversal);
    }

    public void OnPostorder() {
        view.stopAnimation();
        List<Integer> traversal = tree.postorder();
        view.setStatus("Postorder traversal: " + traversal.toString());
        view.animateTraversal(traversal);
```

```
    }

    public void InvalidInput(String msg) {
        view.setStatus(msg);
    }
}
```