BINARY SEARCH TREE

CODE

/\*

This is the javafx code for

BINARY SEARCH TREE

By Riya and Khushi

\*/

// Package name

package application;

// Importing all the required classes from java and javafx library

import java.util.Collection;

import java.util.List;

import javafx.application.Application;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.layout.Background;

import javafx.scene.layout.BackgroundFill;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.HBox;

import javafx.scene.layout.Pane;

import javafx.scene.paint.Color;

import javafx.scene.shape.Circle;

import javafx.scene.shape.Line;

import javafx.scene.text.Font;

import javafx.scene.text.FontPosture;

import javafx.scene.text.FontWeight;

import javafx.scene.text.Text;

import javafx.stage.Stage;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*VIEW\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class BTView extends Pane {

private BinarySearchTree<Integer> tree = new BinarySearchTree<>();

private double radius = 30; // Radius of each Treenode

private double vGap = 50; // Distance between two levels in a tree

// Parameterized constructor for BTView class

BTView(BinarySearchTree<Integer> tree) {

this.tree = tree; // Using this pointer to access private data member

setStatus("Tree is empty"); // Default statement informing the tree has not been formed yet

setStatus1("Height: 0"); // Default statement displaying the height of an empty tree as zero

setStatus2("Vertices: 0"); // Default statement displaying the vertices in an empty tree as zero

}

// Function defined to set the properties of the message displayed with every button action performed

public void setStatus(String msg) {

Text t = new Text (20, 20, msg);

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 20)); // Font properties

t.setFill(Color.***DARKGREY***); // Font color

getChildren().add(t);

}

// Function defined to set the properties of the Height displayed

public void setStatus1(String msg) {

Text t = new Text ((getWidth()/ 2) - 150, getHeight() - 50, msg);

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 25)); // Font properties

t.setFill(Color.***DARKGREY***); // Font color

getChildren().add(t);

}

// Function defined to set the properties of the Vertices displayed

public void setStatus2(String msg) {

Text t = new Text ((getWidth()/ 2) + 75, getHeight() - 50, msg);

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 25)); // Font properties

t.setFill(Color.***DARKGREY***); // Font color

getChildren().add(t);

}

//public call to access the private displayTree function

public void displayTree() {

this.getChildren().clear(); // Clear the pane

if (tree.getRoot() != null) { // Display tree recursively

displayTree(tree.getRoot(), getWidth() / 2, vGap, getWidth() / 4);

// Reduces the distance between two levels of the tree as height increases to keep the tree within the window

}

}

// Display a subtree rooted at position (x, y)

private void displayTree(TreeNode<Integer> root, double x, double y, double hGap) {

if (root.left != null) {

// Draw a line to the left node

Line ll = new Line(x - hGap, y + vGap, x, y); //ll = left line

// Setting the properties of the line

ll.setStroke(Color.***GREEN***); // Color of the line

ll.setStrokeWidth(3); // Width of the line

getChildren().add(ll);

// Draw the left subtree recursively

displayTree(root.left, x - hGap, y + vGap, hGap / 2 );

}

if (root.right != null) {

// Draw a line to the right node

Line lr = new Line(x + hGap, y + vGap, x, y); //lr = right line

// Setting the properties of the line

lr.setStroke(Color.***GREEN***); // Color of the line

lr.setStrokeWidth(3); // Width of the line

getChildren().add(lr);

// Draw the right subtree recursively

displayTree(root.right, x + hGap, y + vGap, hGap / 2);

}

// Displaying each node inside a circle

Circle circle = new Circle(x, y, radius);

// Circle properties

circle.setFill(Color.***GREENYELLOW***); //Color inside the circle

circle.setStroke(Color.***GREEN***); //Color of the circumference of the circle

circle.setStrokeWidth(4); //Width of the circumference of the circle

//Placing the valueof each node inside a text object

Text t = new Text(x - 4, y + 4, root.element + "");

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 20)); // Font properties

t.setFill(Color.***BLACK***); // Font color

getChildren().addAll(circle, t);

}

//public call to access the private displayTree function

public void displaySearch(int key) {

this.getChildren().clear(); // Clear the pane

if (tree.getRoot() != null) {

// Display tree recursively

displaySearch(tree.getRoot(), getWidth() / 2, vGap, getWidth() / 4, key);

// Reduces the distance between two levels of the tree as height increases to keep the tree within the window

}

}

//A separate display function is created to highlight the searched element in the tree

private void displaySearch(TreeNode<Integer> root, double x, double y, double hGap, int key) {

if (root.left != null) {

// Draw a line to the left node

Line ll = new Line(x - hGap, y + vGap, x, y); //ll = left line

// Setting the properties of the line

ll.setStroke(Color.***GREEN***); // Color of the line

ll.setStrokeWidth(3); // Width of the line

getChildren().add(ll);

// Draw the left subtree recursively

displayTree(root.left, x - hGap, y + vGap, hGap / 2 );

}

if (root.right != null) {

// Draw a line to the right node

Line lr = new Line(x + hGap, y + vGap, x, y); //lr = right line

// Setting the properties of the line

lr.setStroke(Color.***GREEN***); // Color of the line

lr.setStrokeWidth(3); // Width of the line

getChildren().add(lr);

// Draw the right subtree recursively

displayTree(root.right, x + hGap, y + vGap, hGap / 2);

}

// Displaying each node inside a circle

Circle circle = new Circle(x, y, radius);

if(root.element != key) {

circle.setFill(Color.***GREENYELLOW***); // Default color inside eah circle

}

else {

circle.setFill(Color.***MEDIUMORCHID***); // Color inside the circle if the element is found

}

circle.setStroke(Color.***GREEN***); //Color of the circumference of the circle

circle.setStrokeWidth(3); //Width of the circumference of the circle

Text t = new Text(x - 4, y + 4, root.element + "");

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 20)); // Font properties

t.setFill(Color.***BLACK***); // Font color

getChildren().addAll(circle, t);

}

//public call to access the private displayPreorder function

public void displayPreorder() {

if (tree.getRoot() != null) {

// Recursive call to display tree

displayPreorder(tree.getRoot());

}

}

// Declaring the string outside the recursive function to prevent re-initialization during recursive call

String s = "Preorder: ";

private void displayPreorder(TreeNode<Integer> root) {

//returns if tree is empty

if (root == null)

return;

//concatenating each element according to preorder

s = s.concat(Integer.*toString*(root.element) + " ");

Text t = new Text (20, 20, s);

//setting font properties

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 20));

t.setFill(Color.***DARKGREY***);

getChildren().add(t);

//recursive calls

displayPreorder(root.left);

displayPreorder(root.right);

}

//public call to access the private displayInorder function

public void displayInorder() {

if (tree.getRoot() != null) {

// Recursive call to display tree

displayInorder(tree.getRoot());

}

}

// Declaring the string outside the recursive function to prevent re-initialization during recursive call

String p = "Inorder: ";

private void displayInorder(TreeNode<Integer> root) {

//returns if tree is empty

if (root == null)

return;

//recursive calls

displayInorder(root.left);

//concatenating each element according to inorder

p = p.concat(Integer.*toString*(root.element) + " ");

Text t = new Text (20, 20, p);

//setting font properties

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 20));

t.setFill(Color.***DARKGREY***);

getChildren().add(t);

//recursive calls

displayInorder(root.right);

}

//public call to access the private displayPostorder function

public void displayPostorder() {

if (tree.getRoot() != null) {

// Recursive call to display tree

displayPostorder(tree.getRoot());

}

}

// Declaring the string outside the recursive function to prevent re-initialization during recursive call

String q = "Postorder: ";

private void displayPostorder(TreeNode<Integer> root) {

//returns if tree is empty

if (root == null)

return;

//recursive calls

displayPostorder(root.left);

displayPostorder(root.right);

//concatenating each element according to inorder

q = q.concat(Integer.*toString*(root.element) + " ");

Text t = new Text (20, 20, q);

//setting font properties

t.setFont(Font.*font* ("Times New Roman", *FontWeight*.***BOLD***, 20));

t.setFill(Color.***DARKGREY***);

getChildren().add(t);

}

//Function to display height of tree

public void displayHeight() {

int h;

h = tree.height(tree.getRoot());

setStatus1("Height: " + h);

}

//Function to display the number of vertices in the tree

public void displayVertices() {

int v;

v = tree.vertices(tree.getRoot());

setStatus2("Vertices: " + v);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MODEL\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class Main extends Application {

*@Override*

public void start(Stage primaryStage) {

BinarySearchTree<Integer> tree = new BinarySearchTree<>(); // Creating a Tree

BorderPane pane = new BorderPane();

BTView view = new BTView(tree); // Creating a View

pane.setCenter(view);

// create Background

// Background background = new Background(background\_fill);

pane.setStyle("-fx-background-color: #333232;");

TextField tfKey = new TextField(); // Textfield for taking the input from the user

tfKey.setPrefColumnCount(5);

tfKey.setAlignment(*Pos*.***BASELINE\_RIGHT***);

//Display properties of the textfield

tfKey.setStyle("-fx-font-size: 1.5em ; -fx-text-fill: DARKGREY; -fx-background-color: #524D4D;");

Button btInsert = new Button("Insert"); //Insert button

Button btDelete = new Button("Delete"); //Delete button

Button btSearch = new Button("Search"); //Search button

Button btPreorder = new Button("Preorder"); //Preorder button

Button btInorder = new Button("Inorder"); //Inorder button

Button btPostorder = new Button("Postorder"); //Postorder button

//Insert button properties

btInsert.setMinWidth(100);

btInsert.setMinHeight(40);

btInsert.setStyle("-fx-font-size: 2em ; -fx-text-fill: DARKGREY ; -fx-background-color: #524D4D;");

//Delete button properties

btDelete.setMinWidth(100);

btDelete.setMinHeight(40);

btDelete.setStyle("-fx-font-size: 2em ; -fx-text-fill: DARKGREY ; -fx-background-color: #524D4D;");

//Search button properties

btSearch.setMinWidth(100);

btSearch.setMinHeight(40);

btSearch.setStyle("-fx-font-size: 2em; -fx-text-fill: DARKGREY ; -fx-background-color: #524D4D; ");

//Preorder button properties

btPreorder.setMinWidth(100);

btPreorder.setMinHeight(40);

btPreorder.setStyle("-fx-font-size: 2em; -fx-text-fill: DARKGREY ; -fx-background-color: #524D4D; ");

//Inorder button properties

btInorder.setMinWidth(100);

btInorder.setMinHeight(40);

btInorder.setStyle("-fx-font-size: 2em; -fx-text-fill: DARKGREY ; -fx-background-color: #524D4D; ");

//Postorder button properties

btPostorder.setMinWidth(100);

btPostorder.setMinHeight(40);

btPostorder.setStyle("-fx-font-size: 2em; -fx-text-fill: DARKGREY ; -fx-background-color: #524D4D; ");

//Creating an HBox to accomodate the name of the application

HBox hBox1 = new HBox(20);

hBox1.setPrefSize(100,60); // Size of the HBox

hBox1.setBackground(new Background(new BackgroundFill(Color.***BLACK***,null,null))); //Background properties of the HBox

//Creating a Label to hold the name of our application

Label label = new Label("BINARY SEARCH TREE ");

//Setting font properties

label.setTextFill(Color.*web*("#979595", 0.8));

Font f=Font.*font*("Times New Roman", *FontWeight*.***BOLD***,*FontPosture*.***ITALIC***, 25);

label.setFont(f);

//Placing the HBox to the bottom of the pane

hBox1.getChildren().addAll(label);

hBox1.setAlignment(*Pos*.***CENTER***);

pane.setBottom(hBox1);

//Creating a separate HBox to accomodate all the buttons and textfield

HBox hBox = new HBox(20);

hBox.setPrefSize(100,60); // Size of the HBox

hBox.setBackground(new Background(new BackgroundFill(Color.***BLACK***,null,null)));//Background properties of the HBox

//Creating a label to define the purpose of the textfield

Label label1 = new Label("Enter a key: ");

//Lable properties

label1.setTextFill(Color.*web*("#979595", 0.8));

Font f1=Font.*font*("Times New Roman", *FontWeight*.***BOLD***, 25);

label1.setFont(f1);

//Inserting all the elements of the Hbox to the pane

//Placing the HBox to the top of the pane

hBox.getChildren().addAll(label1, tfKey, btInsert, btDelete, btSearch, btPreorder, btInorder, btPostorder);

hBox.setAlignment(*Pos*.***CENTER***);

pane.setTop(hBox);

//Event handler that comes in action when Insert button is clicked

btInsert.setOnAction(e -> {

int key = Integer.*parseInt*(tfKey.getText());

if (tree.search(key)) { // key is in the tree already

view.displayTree(); //displays tree

view.setStatus(key + " is already in the tree");

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

} else {

tree.insert(key); // Insert a new key

view.displayTree(); //displays tree

view.setStatus(key + " is inserted in the tree");

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

}

});

//Event handler that comes in action when Delete button is clicked

btDelete.setOnAction(e -> {

int key = Integer.*parseInt*(tfKey.getText());

if (!tree.search(key)) { // key is not in the tree

view.displayTree(); //displays tree

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

view.setStatus(key + " is not in the tree");

} else {

tree.delete(key); // Delete a key

view.displayTree(); //displays tree

view.setStatus(key + " is deleted from the tree");

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

}

});

//Event handler that comes in action when Search button is clicked

btSearch.setOnAction(e -> {

int key = Integer.*parseInt*(tfKey.getText());

if (!tree.search(key)) { // key is not in the tree

view.displayTree(); //displays tree

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

view.setStatus(key + " is not in the tree");

} else {

view.displayTree(); //displays tree

view.displaySearch(key); // Search a key

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

view.setStatus(key + " is in the tree");

}

});

//Event handler that comes in action when Preorder button is clicked

btPreorder.setOnAction(e -> {

view.displayTree(); //displays tree

// view.setStatus("Preorder is");

view.displayPreorder(); //displays all node values in preorder

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

});

//Event handler that comes in action when Inorder button is clicked

btInorder.setOnAction(e -> {

view.displayTree(); //displays tree

//view.setStatus("Inorder is");

view.displayInorder(); //displays all node values in inorder

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

});

//Event handler that comes in action when Postorder button is clicked

btPostorder.setOnAction(e -> {

view.displayTree(); //displays tree

//view.setStatus("Postorder is");

view.displayPostorder(); //displays all node values in postorder

view.displayHeight(); //displays height of tree

view.displayVertices(); //displays number of vertices of tree

});

// Create a scene and place the pane in the stage

Scene scene = new Scene(pane, 450, 250);

primaryStage.setTitle("Binary Search Tree"); //Title of the applcation

primaryStage.setScene(scene);

primaryStage.show();

}

//Launching the application

public static void main(String[] args) {

*launch*(args);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CONTROLLER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class BinarySearchTree<E extends Comparable<E>> implements Tree<E> {

protected TreeNode<E> root;

protected int size = 0;

//Empty function

public BinarySearchTree() {

}

//Adding nodes to an element array

public BinarySearchTree(E[] objects) {

for (int i = 0; i < objects.length; i++)

add(objects[i]);

}

//Search Function

*@Override*

public boolean search(E e) {

TreeNode<E> current = root; // Start from the root

//This search algorithm does not require traversal of the entire tree

while (current != null) {

if (e.compareTo(current.element) < 0) {

current = current.left; //if compareTo returns <0 then element required closer to left node

} else if (e.compareTo(current.element) > 0) {

current = current.right; //if compareTo returns >0 then element required closer to left node

} else // element matches current.element

return true; // Element is found

}

return false;

}

//Height Function

public int height(TreeNode<E> r) {

if (r == null)

return 0;

else

{

//recursively compute the depth of each subtree

int lDepth = height(r.left);

int rDepth = height(r.right);

//traversing the larger depth

if (lDepth > rDepth)

return (lDepth + 1);

else

return (rDepth + 1);

}

}

//Function for calculating Number of Vertices

public int vertices(TreeNode<E> r)

{

if (r == null)

return 0;

else

{ return height(r.left) + height(r.right) + 1; }

}

//Insert Function

*@Override*

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 successfully

}

protected TreeNode<E> createNewNode(E e) {

return new TreeNode<>(e);

}

*@Override* /\*\* Inorder traversal from the root \*/

public void inorder() {

inorder(root);

}

/\*\* Inorder traversal from a subtree \*/

protected void inorder(TreeNode<E> root) {

if (root == null)

return;

inorder(root.left);

System.***out***.print(root.element + " ");

inorder(root.right);

}

*@Override* /\*\* Postorder traversal from the root \*/

public void postorder() {

postorder(root);

}

/\*\* Postorder traversal from a subtree \*/

protected void postorder(TreeNode<E> root) {

if (root == null)

return;

postorder(root.left);

postorder(root.right);

System.***out***.print(root.element + " ");

}

*@Override* /\*\* Preorder traversal from the root \*/

public void preorder() {

preorder(root);

}

/\*\* Preorder traversal from a subtree \*/

protected void preorder(TreeNode<E> root) {

if (root == null)

return;

System.***out***.print(root.element + " ");

preorder(root.left);

preorder(root.right);

}

*@Override* /\*\* 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 List<TreeNode<E>> path(E e) {

List<TreeNode<E>> list = new java.util.ArrayList<>();

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;

}

*@Override*

public boolean delete(E e) {

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;

}

if (current == null)

return false; // Element is not in the tree

// Case 1: current has no left child

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 deleted successfully

}

*@Override* /\*\* Obtain an iterator. Use in order. \*/

public java.util.Iterator<E> iterator() {

return new InorderIterator();

}

// Inner class InorderIterator

private class InorderIterator implements java.util.Iterator<E> {

// Store the elements in a list

private java.util.ArrayList<E> list = new java.util.ArrayList<>();

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);

}

*@Override* /\*\* More elements for traversing? \*/

public boolean hasNext() {

if (current < list.size())

return true;

return false;

}

*@Override* /\*\* Get the current element and move to the next \*/

public E next() {

return list.get(current++);

}

*@Override* // Remove the element returned by the last next()

public void remove() {

if (current == 0) // next() has not been called yet

throw new IllegalStateException();

delete(list.get(--current));

list.clear(); // Clear the list

inorder(); // Rebuild the list

}

}

*@Override* /\*\* Remove all elements from the tree \*/

public void clear() {

root = null;

size = 0;

}

*@Override*

//default function created to implement all the methods of the abstract class

public Object[] toArray() {

// TODO Auto-generated method stub

return null;

}

*@Override*

//default function created to implement all the methods of the abstract class

public <T> T[] toArray(T[] a) {

// TODO Auto-generated method stub

return null;

}

*@Override*

//default function created to implement all the methods of the abstract class

public boolean containsAll(Collection<?> c) {

// TODO Auto-generated method stub

return false;

}

*@Override*

//default function created to implement all the methods of the abstract class

public boolean addAll(Collection<? extends E> c) {

// TODO Auto-generated method stub

return false;

}

*@Override*

//default function created to implement all the methods of the abstract class

public boolean removeAll(Collection<?> c) {

// TODO Auto-generated method stub

return false;

}

*@Override*

//default function created to implement all the methods of the abstract class

public boolean retainAll(Collection<?> c) {

// TODO Auto-generated method stub

return false;

}

}

interface Tree<E> extends Collection<E> {

/\*\* Return true if the element is in the tree \*/

public boolean search(E e);

/\*\* Traversing to calculate the height of the tree \*\*/

public default void height(E e) {

}

/\*\*Calculating vertices\*\*/

public default void vertices(E e) {

}

/\*\*

\* Insert element e 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);

/\*\* Get the number of elements in the tree \*/

public int getSize();

/\*\* Inorder traversal from the root \*/

public default void inorder() {

}

/\*\* Postorder traversal from the root \*/

public default void postorder() {

}

/\*\* Preorder traversal from the root \*/

public default void preorder() {

}

*@Override* // Returns true if the tree is empty

public default boolean isEmpty() {

return this.size() == 0;

}

//Functions for accessing private data members

*@Override*

public default boolean contains(Object e) {

return search((E) e);

}

//Functions for accessing private data members

*@Override*

public default boolean add(E e) {

return insert(e);

}

//Functions for accessing private data members

*@Override*

public default boolean remove(Object e) {

return delete((E) e);

}

//Functions for accessing private data members

*@Override*

public default int size() {

return getSize();

}

}

class TreeNode<E> {

public E element;

public TreeNode<E> left;

public TreeNode<E> right;

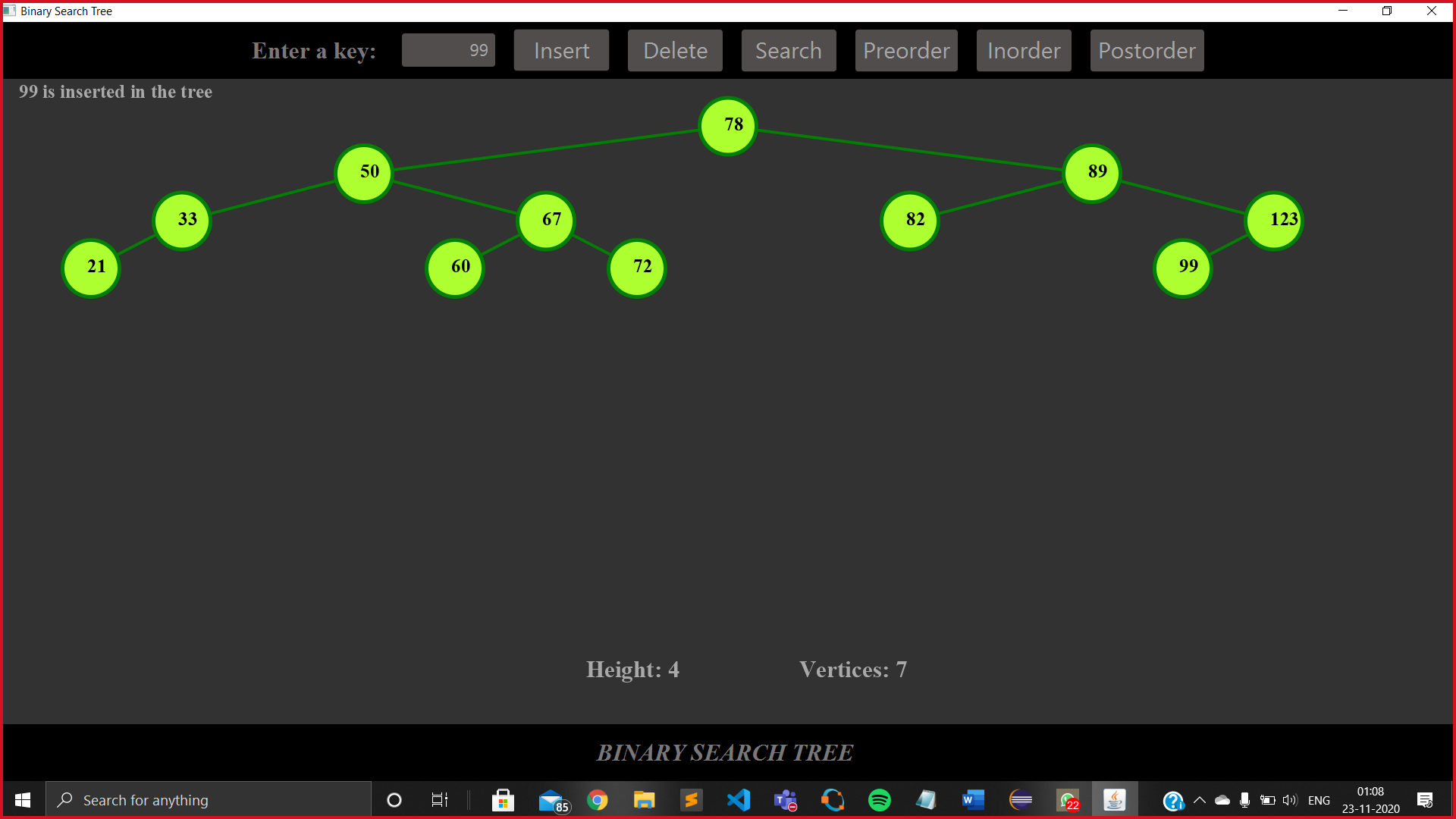
public TreeNode(E e) {

element = e;

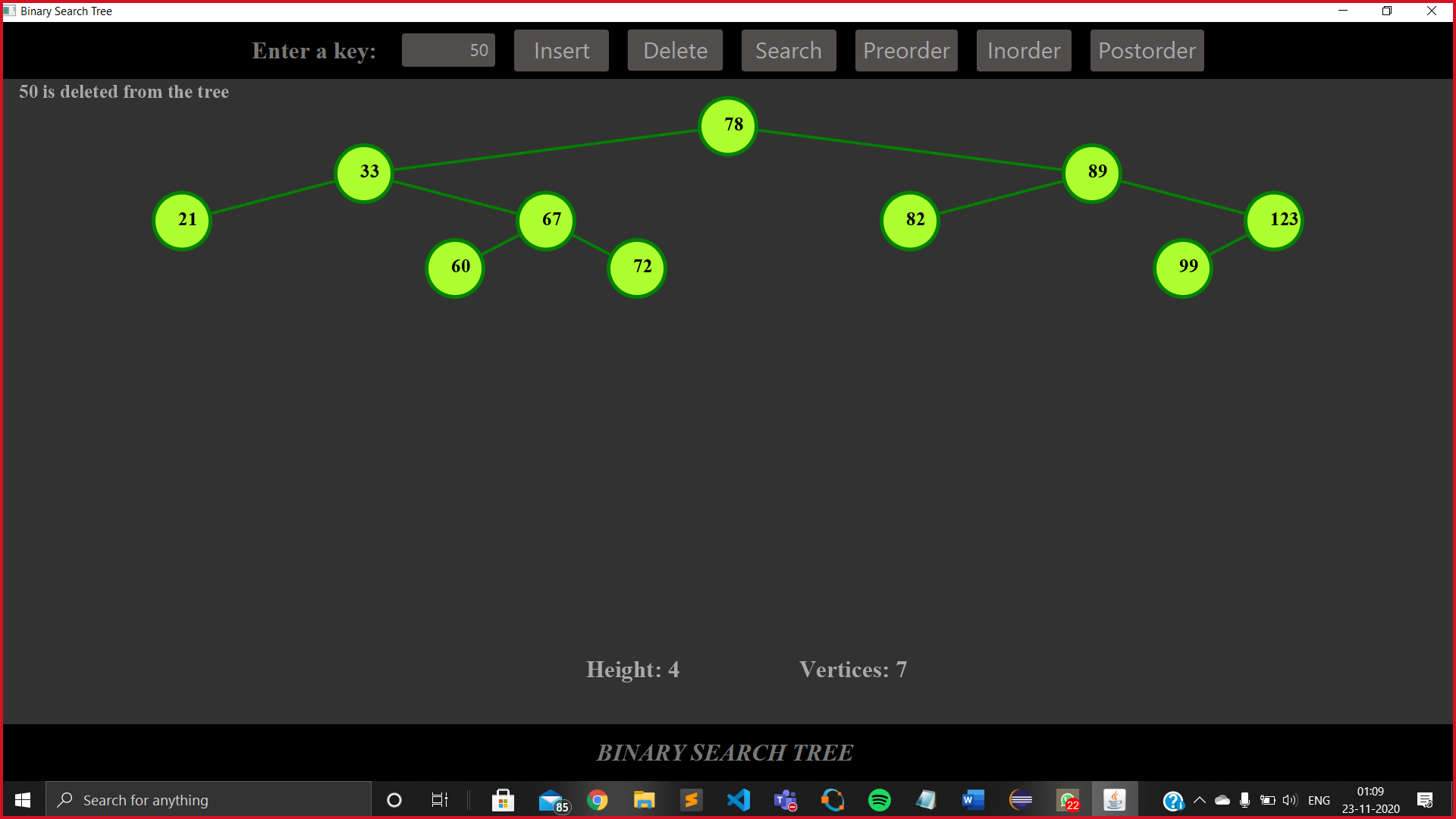
}

OUTPUT

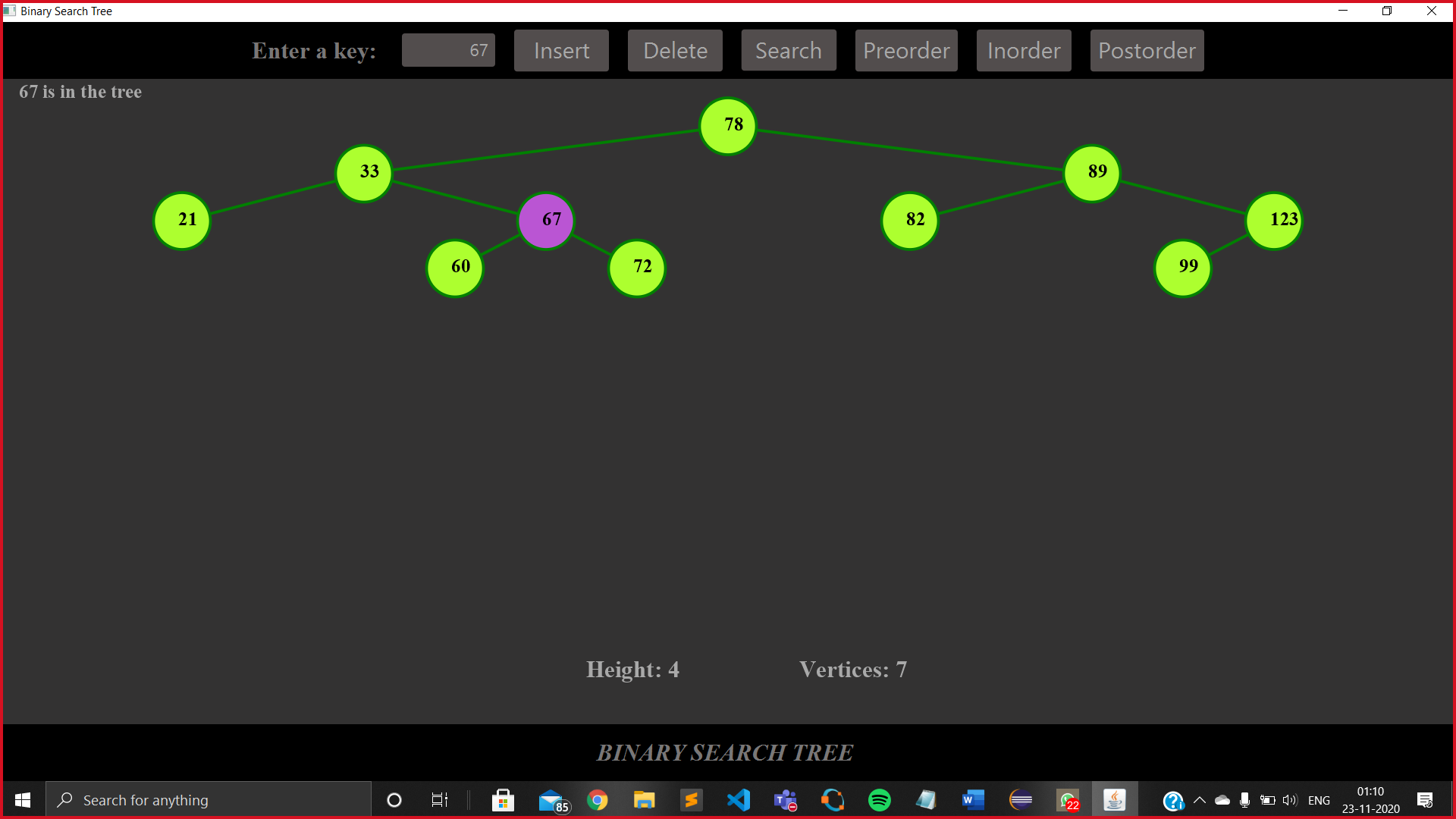
Inserting numbers in the tree and displaying them simultaneously.



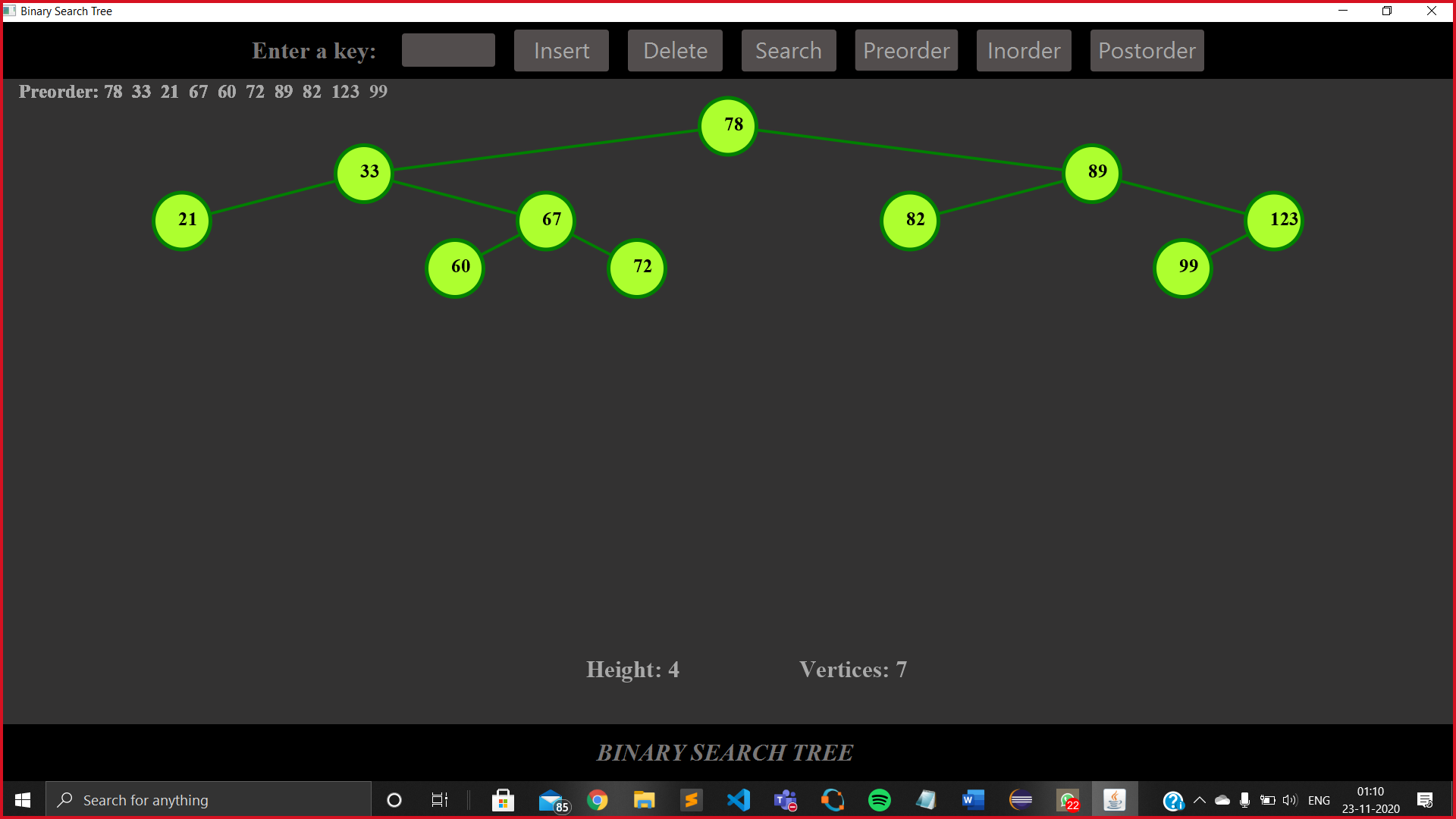
Deleting a number from the tree.



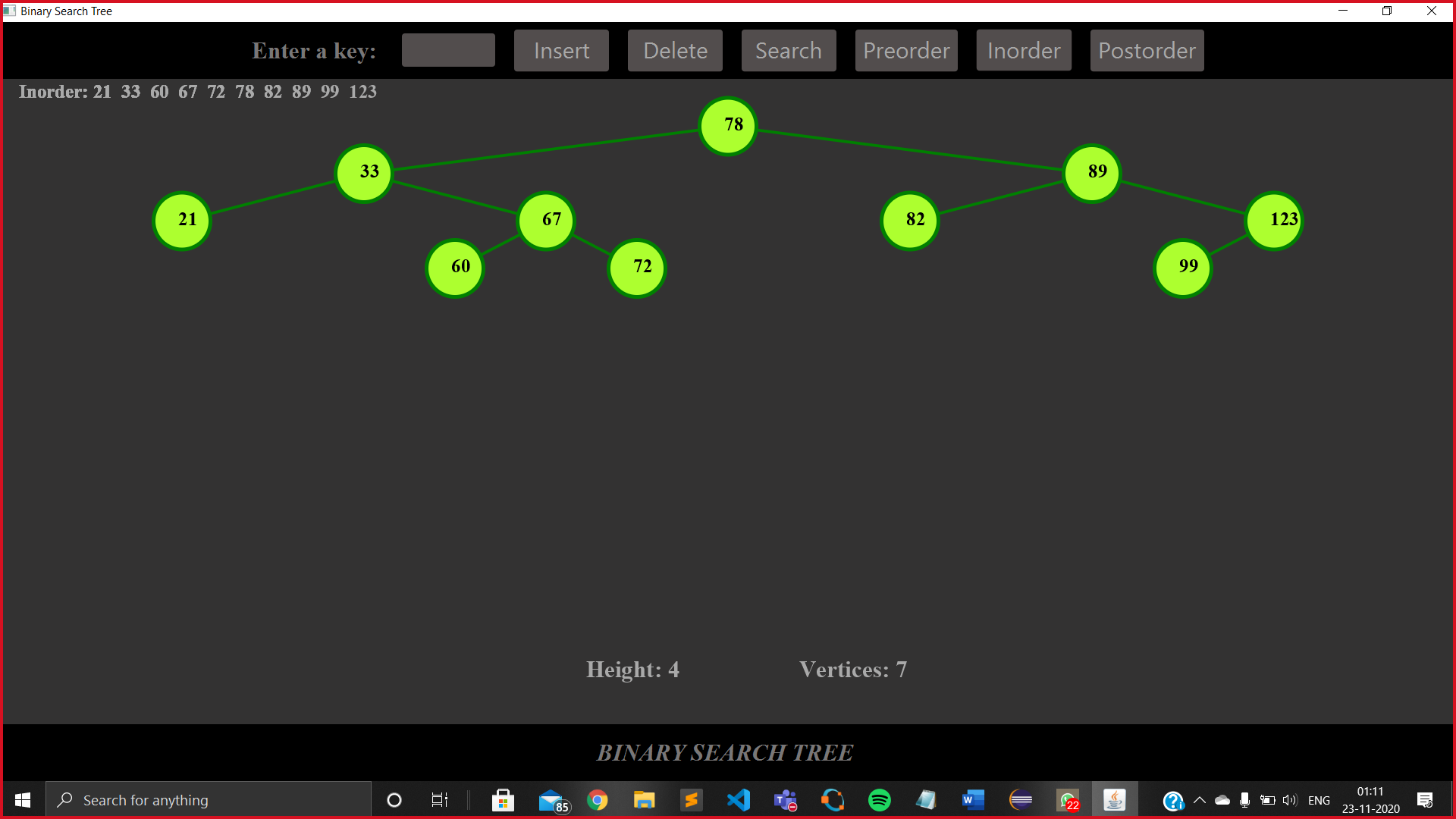
Searching a specific number in the tree and highlighting that number.



Displaying the preorder.



Displaying the inorder.



Displaying the postorder.

