For this project assignment, you will solve a problem based on what you have learnt in this course.

**Instructions**

* Write your name and SNU\_ID in the header of this document.
* Assignment submitted after the due date will not be evaluated and a score of zero will be awarded.
* Plagiarized assignments will also be awarded zero.
* Upload a word version of this document.
* Properly document/comment your code, followed by snapshots of output as desired.

**Submitting this Assignment**

* You will submit (upload) this assignment in Blackboard.
* Email/paper submissions will not be accepted.
* Name this document as Project\_CSD203-2020\_John\_Bill.doc in case the first names of group members are John and Bill respectively.

**Grading Criteria**

**This assignment has 13 points (with weightage of 13% in your overall 100 points). Points will be awarded as follows:**

1. Functionality – **10 points**
2. Look and Feel of node creation, deletion and searching implementations – **3 points**

**Project Problem**

Write a java program to create an **m-Way Search Tree or B-Tree**(where **m** may be either of the 3, 4, or 5)that will make use of several JavaFX components, event handling, graphics, and Java Collections Framework to implement. GUI must contain buttons to perform following operations:

1. Insert - to insert a node (element) into the tree (3 Marks)
2. Delete - to delete a node from the tree (3 Marks)
3. Find- to search an element in the tree (2 Marks)

Program should keep updating the following details at the bottom of the Frame:

1. height of the tree (1 Marks)
2. number of vertices (1 Marks)

**NOTE: CODE IS PASTED FROM THE NEXT PAGE**

CODE:

B-TREE ALGORITHM CLASSES:

**Operations.Java Class:**

package btree;

public class Operations<T extends Comparable<T>> {

private static final int ROOT\_IS\_BIGGER = 1;

private static final int ROOT\_IS\_SMALLER = -1;

private Node root; // Tree root

private int size; // The number of tree elements

private boolean flag; // Tracks if the last element was added correctly or not.

private int vertices = 0;

public Operations() {

this.root = new Node();

this.size = 0;

}

public Node getRoot() {

return this.root;

}

/\*\*

\* @return true, the tree is empty, otherwise false

\*/

public boolean isEmpty() {

if (root == null) return true;

return root.getLeftElement() == null;

}

/\*\*

\* Check if the given element is in the tree

\*

\* @param element the element to check

\* @return true, if the element is found, otherwise false

\*/

public boolean contains(T element) {

return search(element);

}

/\*\*

\* @return number of elements in the tree

\*/

public int size() {

return size;

}

/\*\*

\* Adds a new element to the tree, keeping it balanced

\*

\* @param element - element to add

\*/

public void add(T element) {

flag = false;

if (root == null || root.getLeftElement() == null) {

flag = true;

if (root == null) {

root = new Node();

}

root.setLeftElement(element);

} else {

Node newRoot = add(root, element);

if (newRoot != null) {

root = newRoot;

}

}

if (flag) size++;

}

/\*\*

\* @param current node to add to

\* @param element - element to add

\*/

private Node add(Node current, T element) {

Node newParent = null; // Node to be added

// We are not yet at the deepest level

if (!current.isLeaf()) {

Node newNode;

// Element already exists

if (current.leftElement.compareTo(element) == 0 || (current.is3Node() && current.rightElement.compareTo(element) == 0)) {

}

// newNode < left element

else if (current.leftElement.compareTo(element) == ROOT\_IS\_BIGGER) {

newNode = add(current.leftChild, element);

// newNode comes from the left branch

if (newNode != null) {

// newNode < than current.left

if (current.is2Node()) {

current.rightElement = current.leftElement; // Move the current left element to the right

current.leftElement = newNode.leftElement;

current.rightChild = current.middleChild;

current.middleChild = newNode.middleChild;

current.leftChild = newNode.leftChild;

}

// We have a new division, so the current element on the left will rise

else {

// Copy the right side of the subtree

Node rightCopy = new Node(current.rightElement, null, current.middleChild, current.rightChild);

// Create a new "structure" by inserting the right side

newParent = new Node(current.leftElement, null, newNode, rightCopy);

}

}

}

// newNode is > left and < right

else if (current.is2Node() || (current.is3Node() && current.rightElement.compareTo(element) == ROOT\_IS\_BIGGER)) {

newNode = add(current.middleChild, element);

// New division

if (newNode != null) {

// The right element is empty, so we can set newNode on the left, and the existing left element on the right

if (current.is2Node()) {

current.rightElement = newNode.leftElement;

current.rightChild = newNode.middleChild;

current.middleChild = newNode.leftChild;

}

// Another case where we have to split again

else {

Node left = new Node(current.leftElement, null, current.leftChild, newNode.leftChild);

Node mid = new Node(current.rightElement, null, newNode.middleChild, current.rightChild);

newParent = new Node(newNode.leftElement, null, left, mid);

}

}

}

// newNode is larger than the right element

else if (current.is3Node() && current.rightElement.compareTo(element) == ROOT\_IS\_SMALLER) {

newNode = add(current.rightChild, element);

// Divide -> the right element rises

if (newNode != null) {

Node leftCopy = new Node(current.leftElement, null, current.leftChild, current.middleChild);

newParent = new Node(current.rightElement, null, leftCopy, newNode);

}

}

}

// We are at the deepest level

else {

flag = true;

// Element already exists

if (current.leftElement.compareTo(element) == 0 || (current.is3Node() && current.rightElement.compareTo(element) == 0)) {

flag = false;

}

// The case when there is no right element

else if (current.is2Node()) {

// If the current left element is larger than newNode, we move the left element to the right

if (current.leftElement.compareTo(element) == ROOT\_IS\_BIGGER) {

current.rightElement = current.leftElement;

current.leftElement = element;

}

// If newNode is larger, we add it to the right

else if (current.leftElement.compareTo(element) == ROOT\_IS\_SMALLER) current.rightElement = element;

}

// The case when the node has 2 elements, and we want to add another one. To do this, we share the node

else newParent = split(current, element);

}

return newParent;

}

/\*\*

\* The method creates a new node structure that will be attached at the bottom of the add() method

\*

\* @param current - the node where the separation occurs

\* @param element - element to insert

\* @return two-node structure with a nonzero left and middle node

\*/

private Node split(Node current, T element) {

Node newParent = null;

// The left element is larger, so it will rise, allowing newParent to stand on the left

if (current.leftElement.compareTo(element) == ROOT\_IS\_BIGGER) {

Node<T> left = new Node<>(element, null);

Node right = new Node(current.rightElement, null);

newParent = new Node(current.leftElement, null, left, right);

} else if (current.leftElement.compareTo(element) == ROOT\_IS\_SMALLER) {

Node left = new Node(current.leftElement, null);

// newParent is greater than the current on the right and smaller than the right. newParent rises.

if (current.rightElement.compareTo(element) == ROOT\_IS\_BIGGER) {

Node right = new Node(current.rightElement, null);

newParent = new Node(element, null, left, right);

}

// newParent is the largest, so the current right element is raised

else {

Node<T> right = new Node<>(element, null);

newParent = new Node(current.rightElement, null, left, right);

}

}

return newParent;

}

/\*\*

\* Method for removing an element from the tree

\*

\* @param element - element to remove

\* @return true, if the element was removed, otherwise false

\*/

public boolean remove(T element) {

// Reduce the number of levels at the beginning

this.size--;

boolean ifRemoved = remove(root, element);

root.reBalance();

// If you deleted the last element of the tree

if (root.getLeftElement() == null) root = null;

// If the element could not be deleted, then increase the number of levels

if (!ifRemoved) this.size++;

return ifRemoved;

}

/\*\*

\* @param current - node to be deleted

\* @param element - element to be deleted

\* @return true, if the element was deleted, otherwise false

\*/

private boolean remove(Node current, T element) {

boolean ifRemoved = true;

// The case when we are at the deepest level of the tree, but we did not find the element (it does not exist)

if (current == null) {

ifRemoved = false;

return false;

}

// Recursive case, we still find the element to delete

else {

if (!current.getLeftElement().equals(element)) {

// If there is no element on the right or the element to be deleted is smaller than the right element

if (current.getRightElement() == null || current.getRightElement().compareTo(element) == ROOT\_IS\_BIGGER) {

// The left element is larger than the element to be deleted, so we go through the left child element

if (current.getLeftElement().compareTo(element) == ROOT\_IS\_BIGGER) {

ifRemoved = remove(current.leftChild, element);

}

// Otherwise -> try to remove the middle child

else {

ifRemoved = remove(current.middleChild, element);

}

} else {

// If the element to be deleted is not equal to the desired element, we pass the right child

if (!current.getRightElement().equals(element)) {

ifRemoved = remove(current.rightChild, element);

}

// Otherwise, we found an element

else {

// \*\*\* Situation 1 \*\*\*

// The element is equal to the right element of the sheet, so we just delete it

if (current.isLeaf()) {

current.setRightElement(null);

}

// \*\*\* Situation 2 \*\*\*

// We found the element, but it is not in the sheet

else {

// We get the min element of the right branch,

// delete it from the current position and place it where we found the element to delete.

T replacement = (T) current.getRightNode().replaceMin();

current.setRightElement(replacement);

}

}

}

}

// The left element is equal to the element to be deleted.

else {

// \*\*\* Situation 1 \*\*\*

if (current.isLeaf()) {

// The left element, the element to delete, is replaced by the right element

if (current.getRightElement() != null) {

current.setLeftElement(current.getRightElement());

current.setRightElement(null);

}

// If there is no element on the right, then balancing is required

else {

current.setLeftElement(null); // Release the node

return true;

}

}

// \*\*\* Situation 2 \*\*\*

else {

// Move the "max" element of the left branch, where we found the element

T replacement = (T) current.getLeftNode().replaceMax();

current.setLeftElement(replacement);

}

}

}

// The lower level must be balanced

if (!current.isBalanced()) {

current.reBalance();

} else if (!current.isLeaf()) {

boolean isBalanced = false;

while (!isBalanced) {

if (current.getRightNode() == null) {

// A critical case of situation 2 for the left child

if (current.getLeftNode().isLeaf() && !current.getMidNode().isLeaf()) {

T replacement = (T) current.getMidNode().replaceMin();

T tempLeft = (T) current.getLeftElement();

current.setLeftElement(replacement);

add(tempLeft);

}

// A critical case of situation 2 for the right child

else if (!current.getLeftNode().isLeaf() && current.getMidNode().isLeaf()) {

if (current.getRightElement() == null) {

T replacement = (T) current.getLeftNode().replaceMax();

T tempLeft = (T) current.getLeftElement();

current.setLeftElement(replacement);

add(tempLeft);

}

}

}

if (current.getRightNode() != null) {

if (current.getMidNode().isLeaf() && !current.getRightNode().isLeaf()) {

current.getRightNode().reBalance();

}

if (current.getMidNode().isLeaf() && !current.getRightNode().isLeaf()) {

T replacement = (T) current.getRightNode().replaceMin();

T tempRight = (T) current.getRightElement();

current.setRightElement(replacement);

add(tempRight);

} else {

isBalanced = true;

}

}

if (current.isBalanced()) isBalanced = true;

}

}

return ifRemoved;

}

/\*\*

\* Method for removing all elements from a tree

\*/

public void clear() {

this.size = 0;

this.root = null;

}

/\*\*

\* Method for finding an element in a tree

\*

\* @param element - element to find

\* @return true, if the element was found, otherwise false

\*/

public boolean search(T element) {

if (root == null) {

return false;

} else {

return search(root, element);

}

}

private boolean search(Node current, T element) {

boolean ifFound = false;

if (current != null) {

// In the trivial case -> found an element

if (current.leftElement != null && current.leftElement.equals(element)) {

ifFound = true;

}

// Otherwise -> not yet at the deepest level

else {

// Search element equals right element

if (current.rightElement != null && current.rightElement.equals(element)) {

ifFound = true;

}

// Otherwise -> recursive calls

else {

if (current.leftElement.compareTo(element) == ROOT\_IS\_BIGGER) {

ifFound = search(current.leftChild, element);

} else if (current.rightChild == null || current.rightElement.compareTo(element) == ROOT\_IS\_BIGGER) {

ifFound = search(current.middleChild, element);

} else if (current.rightElement.compareTo(element) == ROOT\_IS\_SMALLER) {

ifFound = search(current.rightChild, element);

} else return false;

}

}

}

return ifFound;

}

/\*\*

\* Method for finding the minimum value

\*

\* @return minimum value, otherwise null

\*/

public T findMin() {

if (isEmpty()) return null;

return findMin(root);

}

private T findMin(Node current) {

// Get the minimum element

if (current.getLeftNode() == null) {

return (T) current.leftElement;

}

// Otherwise -> recursive calls

else {

return findMin(current.getLeftNode());

}

}

/\*\*

\* Method for finding the maximum value

\*

\* @return maximum value, otherwise null

\*/

public T findMax() {

if (isEmpty()) {

return null;

} else {

return findMax(root);

}

}

private T findMax(Node current) {

// Recursive calls

if (current.rightElement != null && current.getRightNode() != null) {

return findMax(current.getRightNode());

} else if (current.getMidNode() != null) {

return findMax(current.getMidNode());

}

// Get the maximum element

if (current.rightElement != null) {

return (T) current.rightElement;

} else {

return (T) current.leftElement;

}

}

public void inOrder() {

if (!isEmpty()) {

inOrder(root);

} else {

System.out.print("The tree is empty...");

}

}

/\*\*

\* Method for displaying tree elements in the order of the method - "in-order"

\*/

private void inOrder(Node current) {

if (current != null) {

if (current.isLeaf()) {

System.out.print(current.getLeftElement().toString() + " ");

if (current.getRightElement() != null) {

System.out.print(current.getRightElement().toString() + " ");

}

} else {

inOrder(current.getLeftNode());

System.out.print(current.getLeftElement().toString() + " ");

inOrder(current.getMidNode());

if (current.getRightElement() != null) {

if (!current.isLeaf()) {

System.out.print(current.getRightElement().toString() + " ");

}

inOrder(current.getRightNode());

}

}

}

}

public void preOrder() {

if (!isEmpty()) {

preOrder(root);

} else {

System.out.print("The tree is empty...");

}

}

/\*\*

\* Method for displaying tree elements in the order of the method - "pre-order"

\*/

private void preOrder(Node current) {

if (current != null) {

System.out.print(current.leftElement.toString() + " ");

preOrder(current.leftChild);

preOrder(current.middleChild);

if (current.rightElement != null) {

System.out.print(current.rightElement.toString() + " ");

preOrder(current.rightChild);

}

}

}

public void postOrder() {

if (!isEmpty()) {

postOrder(root);

} else {

System.out.print("The tree is empty...");

}

}

/\*\*

\* Method for displaying tree elements in the order of the method - "post-order"

\*/

private void postOrder(Node current) {

if (current != null) {

postOrder(current.leftChild);

postOrder(current.middleChild);

System.out.print(current.leftElement.toString() + " ");

if (current.rightElement != null) {

System.out.print(current.rightElement.toString() + " ");

postOrder(current.rightChild);

}

}

}

/\*\*

\* Method to find height of a binary tree!

\*/

private int getHeight(Node curr) {

if (curr != null) {

int ldepth = getHeight(curr.leftChild);

int rdepth = getHeight(curr.rightChild);

if (ldepth > rdepth)

return (ldepth + 1);

else

return (rdepth + 1);

} else {

return 0;

}

}

public int getHeight() {

if (isEmpty()) {

return 0;

} else {

return getHeight(root);

}

}

/\*

\* Method to find number of vertices in the tree

\*/

private void getVertices(Node curr){

if(curr==null)

return;

getVertices(curr.leftChild);

getVertices(curr.middleChild);

getVertices(curr.rightChild);

if(curr==root){

vertices++;

}

if(!curr.isLeaf()) {

if (curr.leftElement != null)

vertices++;

if (curr.middleChild != null)

vertices++;

if (curr.rightElement != null)

vertices++;

}

}

public int getVertices() {

vertices = 0;

getVertices(root);

return vertices;

}

}

**Node.Java Class:**

package btree;

public class Node<T extends Comparable<T>> {

Node leftChild;

Node middleChild;

Node rightChild;

T leftElement;

T rightElement;

public Node() {

this.leftChild = null;

this.middleChild = null;

this.rightChild = null;

this.leftElement = null;

this.rightElement = null;

}

/\*\*

\* Constructor of 3 nodes without specific descendants (null references).

\*/

public Node(T leftElement, T rightElement) {

this.leftElement = leftElement;

this.rightElement = rightElement;

leftChild = null;

middleChild = null;

rightChild = null;

}

/\*\*

\* Constructor of 3 nodes with given left and middle nodes / descendants.

\*/

public Node(T leftElement, T rightElement, Node leftChild, Node middleChild) {

this.leftElement = leftElement;

this.rightElement = rightElement;

this.leftChild = leftChild;

this.middleChild = middleChild;

}

public T getLeftElement() {

return leftElement;

}

public void setLeftElement(T element) {

this.leftElement = element;

}

public T getRightElement() {

return rightElement;

}

public void setRightElement(T element) {

this.rightElement = element;

}

private void setLeftNode(Node left) {

this.leftChild = left;

}

public Node getLeftNode() {

return leftChild;

}

private void setMidNode(Node mid) {

this.middleChild = mid;

}

public Node getMidNode() {

return middleChild;

}

private void setRightNode(Node right) {

this.rightChild = right;

}

public Node getRightNode() {

return rightChild;

}

/\*\*

\* @return true, if we are at the deepest level of a tree, otherwise false

\*/

public boolean isLeaf() {

return leftChild == null && middleChild == null && rightChild == null;

}

/\*\*

\* @return true, if the right node does not exist, otherwise false

\*/

public boolean is2Node() {

return rightElement == null;

}

/\*\*

\* @return true, if the right node exists, otherwise false

\*/

public boolean is3Node() {

return rightElement != null;

}

/\*\*

\* Method for checking if a tree is well balanced

\*

\* @return true if the tree is well balanced, otherwise false

\*/

boolean isBalanced() {

boolean balanced = false;

if (isLeaf()) { // If we are at the deepest level (leaf), it is balanced

balanced = true;

} else if (leftChild.getLeftElement() != null && middleChild.getLeftElement() != null) { // There are two cases: 2 nodes or 3 nodes

if (rightElement != null) { // 3 nodes

if (rightChild.getLeftElement() != null) {

balanced = true;

}

} else { // 2 nodes

balanced = true;

}

}

return balanced;

}

public T replaceMax() {

T max;

/\* Trivial case, we are at the deepest level of the tree \*/

if (isLeaf()) {

if (getRightElement() != null) {

max = getRightElement();

setRightElement(null);

// We are lucky, we do not need to rebalance anything

} else {

max = getLeftElement();

setLeftElement(null);

// At the first stage of the recursive function, rebalancing will occur

}

}

/\* Recursive case, we are not at the deepest level \*/

else {

//

if (getRightElement() != null) {

max = (T) rightChild.replaceMax();

}

// If there is an element on the right, we continue on the right

else {

max = (T) middleChild.replaceMax();

}

}

/\* Keep balance \*/

if (!isBalanced()) {

reBalance();

}

return max;

}

/\*\*

\* @return minimum element

\*/

T replaceMin() {

T min;

/\* Trivial case, we are at the deepest level of the tree \*/

if (isLeaf()) {

min = leftElement;

leftElement = null;

// The element was on the right, we skipped it on the left, and nothing happened here

if (rightElement != null) {

leftElement = rightElement;

rightElement = null;

}

}

/\* A recursive case, until we reach the deepest level, we always go down to the left \*/

else {

min = (T) leftChild.replaceMin();

}

// Keep balance

if (!isBalanced()) {

reBalance();

}

return min;

}

/\*\*

\* Method for maintaining balance by rebalancing the deepest level of the tree starting from the second deepest

\*/

void reBalance() {

while (!isBalanced()) {

/\* Imbalance in the left child \*/

if (getLeftNode().getLeftElement() == null) {

// We put the left element of the current node as the left element of the left child

getLeftNode().setLeftElement(getLeftElement());

// Now we replace the left element of the middle descendant as the left element of the current node

setLeftElement((T) getMidNode().getLeftElement());

// If the right element on the middle child exists, we move it to the left

if (getMidNode().getRightElement() != null) {

getMidNode().setLeftElement(getMidNode().getRightElement());

getMidNode().setRightElement(null);

}

// Otherwise, we will make the middle descendant "empty", so the next iteration can resolve this situation,

// if not, then the critical case begins

else {

getMidNode().setLeftElement(null);

}

}

/\* Imbalance in the right child \*/

else if (getMidNode().getLeftElement() == null) {

// Critical case, each node (child) of the deepest level

// has only one element, the algorithm will have to perform balancing from a higher tree level

if (getRightElement() == null) {

if (getLeftNode().getLeftElement() != null && getLeftNode().getRightElement() == null && getMidNode().getLeftElement() == null) {

setRightElement(getLeftElement());

setLeftElement((T) getLeftNode().getLeftElement());

// We delete current descendants

setLeftNode(null);

setMidNode(null);

setRightNode(null);

} else {

getMidNode().setLeftElement(getLeftElement());

if (getLeftNode().getRightElement() == null) {

setLeftElement((T) getLeftNode().getLeftElement());

getLeftNode().setLeftElement(null);

} else {

setLeftElement((T) getLeftNode().getRightElement());

getLeftNode().setRightElement(null);

}

if (getLeftNode().getLeftElement() == null && getMidNode().getLeftElement() == null) {

setLeftNode(null);

setMidNode(null);

setRightNode(null);

}

}

} else {

// We put the right element of the current node as the left element of the middle child

getMidNode().setLeftElement(getRightElement());

// We put the left element of the right child as the right element of the current node

setRightElement((T) getRightNode().getLeftElement());

// If the right child in which we took the last element

// has the right element, we move it to the left of the same child element.

if (getRightNode().getRightElement() != null) {

getRightNode().setLeftElement(getRightNode().getRightElement());

getRightNode().setRightElement(null);

}

// Otherwise, we will make the right child "empty"

else {

getRightNode().setLeftElement(null);

}

}

}

/\* Imbalance on the right \*/

else if (getRightElement() != null && getRightNode().getLeftElement() == null) {

// \*\*\* Situation 1 \*\*\*

// The middle child exists, so we shift the elements to the right

if (getMidNode().getRightElement() != null) {

getRightNode().setLeftElement(getRightElement());

setRightElement((T) getMidNode().getRightElement());

getMidNode().setRightElement(null);

}

// \*\*\* Situation 2 \*\*\*

// The middle child has only the left element,

// then we need to put the right element of the current node as the right element of the middle child.

else {

getMidNode().setRightElement(getRightElement());

setRightElement(null);

}

}

}

}

}

JAVAFX CLASSES/CODE:

**Main.Java Class:**

package sample;

import btree.Operations;

import javafx.animation.KeyFrame;

import javafx.animation.KeyValue;

import javafx.animation.Timeline;

import javafx.application.Application;

import javafx.beans.property.ObjectProperty;

import javafx.beans.property.SimpleObjectProperty;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.AnchorPane;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.HBox;

import javafx.scene.layout.VBox;

import javafx.scene.paint.Color;

import javafx.scene.text.TextAlignment;

import javafx.stage.Stage;

import javafx.geometry.Insets;

import javafx.util.Duration;

public class Main extends Application {

//Initializing necessary variables

private Operations<Integer> btree;

private TextField addInput = new TextField();

private TreeArea BTREEPane;

@Override

public void start(Stage primaryStage) throws Exception {

//primaryStage.setScene(new Scene(root, 300, 275));

btree = new Operations<>();

//double screenWidth = 1300;

//double screenHeight = 600;

primaryStage.setTitle("B Tree Visualizer");

primaryStage.setWidth(1500);

primaryStage.setHeight(800);

//Starting Borderpane

BorderPane root = new BorderPane();

//Horizontal Box containing all buttons and input field

HBox upperArea = new HBox();

upperArea.setPadding(new Insets(15, 12, 15, 12));

upperArea.setSpacing(20);

upperArea.setStyle("-fx-background-color: #4e99e3;");

//Horizontal Box containing Height and Vertices

HBox lowerArea = new HBox();

lowerArea.setPadding(new Insets(15, 12, 15, 12));

lowerArea.setSpacing(20);

lowerArea.setStyle("-fx-background-color: #4e99e3;");

//Buttons and input fields

addInput.setText("");

addInput.setPrefWidth(50);

Label lbl = new Label("ENTER NODE : ");

Button insert = new Button("INSERT");

Button delete = new Button("DELETE");

Button reset = new Button("RESET");

Button search = new Button("SEARCH");

lbl.setStyle("-fx-font-weight: bold; -fx-text-fill: white; -fx-font-size: 14");

insert.setStyle("-fx-background-color: slateblue; -fx-text-fill: white; -fx-font-weight: bold;");

delete.setStyle("-fx-background-color: slateblue; -fx-text-fill: white; -fx-font-weight: bold;");

reset.setStyle("-fx-background-color: slateblue; -fx-text-fill: white; -fx-font-weight: bold;");

search.setStyle("-fx-background-color: slateblue; -fx-text-fill: white; -fx-font-weight: bold;");

//Displaying Height and Vertices

Label l = new Label("HEIGHT: ");

//String ss = String.valueOf(BTREEPane.displayHeight(this.btree)) ;

Label txt1 = new Label();

txt1.setPrefWidth(138);

txt1.setText("TREE DOES NOT EXIST!");

Label txt2 = new Label();

txt2.setPrefWidth(40);

txt2.setText("0");

txt1.setAlignment(Pos.CENTER);

txt2.setAlignment(Pos.CENTER);

Label l2 = new Label("VERTICES: ");

l.setStyle("-fx-font-weight: bold; -fx-text-fill: white; -fx-font-size: 14");

txt1.setStyle("-fx-text-fill: black; -fx-font-size: 12; -fx-background-color: white");

txt2.setStyle("-fx-text-fill: black; -fx-font-size: 12; -fx-background-color: white");

l2.setStyle("-fx-font-weight: bold; -fx-text-fill: white; -fx-font-size: 14");

//Adding all elements in UpperArea

upperArea.getChildren().addAll(lbl, addInput, insert, delete, search, reset, l, txt1, l2, txt2);

upperArea.setAlignment(Pos.CENTER);

root.setTop(upperArea);

//setting up the handler functions

insert.setOnMouseClicked(e -> {

insertNode(addInput.getText());

txt1.setPrefWidth(40);

txt1.setText(String.valueOf(btree.getHeight()));

txt2.setText(String.valueOf(btree.getVertices()));

});

delete.setOnMouseClicked(e -> {

deleteNode(addInput.getText());

txt1.setPrefWidth(40);

txt1.setText(String.valueOf(btree.getHeight()));

txt2.setText(String.valueOf(btree.getVertices()));

});

search.setOnMouseClicked(e -> searchNode(addInput.getText()));

reset.setOnMouseClicked(e -> {

resetTree();

txt1.setPrefWidth(138);

txt1.setText("TREE DOES NOT EXIST!");

txt2.setText("0");

});

//AnchorPane anchorPane = new AnchorPane();

//ScrollPane

//ScrollPane sp = new ScrollPane();

ScrollPane sp = new ScrollPane();

//sp.setPrefSize( primaryStage.getHeight(), primaryStage.getWidth());

// System.out.println(primaryStage.getWidth());

BTREEPane = new TreeArea(btree, primaryStage.getWidth() / 2, 80);

//sp.setPrefSize(300, 300);

sp.setContent(BTREEPane);

sp.setFitToWidth(false);

sp.setFitToHeight(false);

// //sp.setPrefSize(1500, 700);

sp.setHbarPolicy(ScrollPane.ScrollBarPolicy.AS\_NEEDED);

sp.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS\_NEEDED);

sp.setPannable(true);

//Tree Viewing Section

//anchorPane.getChildren().addAll(sp);

root.setCenter(sp);

// root.getChildren().addAll(sp);

Scene scene1 = new Scene(root);

primaryStage.setScene(scene1);

primaryStage.show();

}

//method for inserting a new node to our tree

private void insertNode(String s) {

try {

int num = Integer.parseInt(s);

// System.out.println("Node added is " + num);

this.btree.add(num);

// addValue++;

//addInput.setText("1");

// System.out.println("First inorder");

// this.btree.inOrder();

// System.out.println("");

// if (this.btree.isEmpty()) {

//// System.out.println("its empty");

// }

// System.out.println("Getting root node inside insert node main function");

// System.out.println(this.btree.getRoot().getRightElement());

// if (this.btree.isEmpty()) {

// System.out.println("its empty");

// }

BTREEPane.getChildren().clear();

// System.out.println("Checking what children the tree has");

// System.out.println("Root element is : " + this.btree.getRoot().getLeftElement());

// System.out.println("Root, right element is : " + this.btree.getRoot().getRightElement());

// System.out.println("Root, left child is : " + this.btree.getRoot().getLeftNode());

// System.out.println("Root, middle child is : " + this.btree.getRoot().getMidNode());

// System.out.println("Root, right child is : " + this.btree.getRoot().getRightNode());

BTREEPane.makeTree(this.btree);

addInput.setText("");

// System.out.println("Goes here");

} catch (NumberFormatException e) {

// System.out.println("Invalid input format !");

Alert alert = new Alert(Alert.AlertType.ERROR, "Invalid input format !");

alert.show();

}

}

//method for searching a node

private void searchNode(String s) {

try {

int num = Integer.parseInt(s);

// System.out.println("Node searched is " + num);

if (btree.contains(num)) {

// System.out.println("Searching Now");

BTREEPane.searchTree(num);

} else {

Alert alert = new Alert(Alert.AlertType.INFORMATION, "The element does not exist !");

alert.show();

} //BTREEPane.getChildren().clear();

//BTREEPane.makeTree(btree);

addInput.setText("");

} catch (NumberFormatException e) {

// System.out.println("Invalid input format !");

Alert alert = new Alert(Alert.AlertType.ERROR, "Invalid input format !");

alert.show();

}

}

//method for deleting a node

private void deleteNode(String s) {

try {

int num = Integer.parseInt(s);

// System.out.println("Node deleted is " + num);

btree.remove(num);

BTREEPane.getChildren().clear();

BTREEPane.makeTree(btree);

addInput.setText("");

} catch (NumberFormatException e) {

// System.out.println("Invalid input format !");

Alert alert = new Alert(Alert.AlertType.ERROR, "Invalid input format !");

alert.show();

}

}

//method for resetting the tree

private void resetTree() {

try {

// System.out.println("Tree Reset");

btree.clear();

BTREEPane.resetTree();

BTREEPane.getChildren().clear();

addInput.setText("");

} catch (Exception e) {

// System.out.println("Error occurred !");

Alert alert = new Alert(Alert.AlertType.ERROR, "Error occurred !");

alert.show();

}

}

public static void main(String[] args) {

launch(args);

}

}

**TreeArea.Java Class:**

package sample;

import btree.Operations;

import javafx.animation.FillTransition;

import javafx.animation.KeyFrame;

import javafx.animation.KeyValue;

import javafx.animation.Timeline;

import javafx.scene.layout.Pane;

import javafx.scene.shape.Rectangle;

import javafx.scene.paint.Color;

import javafx.scene.text.Font;

import javafx.scene.text.FontWeight;

import javafx.scene.text.Text;

import btree.Node;

import javafx.scene.shape.Line;

import javafx.util.Duration;

import java.awt.\*;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Objects;

public class TreeArea extends Pane {

private Operations<Integer> btree;

private double startX, startY;

private double nodeWidth = 35;

private double searchNode = -1;

private double lenX = 300;

private double lenY = 80;

//constructor for class TreeArea

public TreeArea(Operations<Integer> btree, double startX, double startY) {

//this.setPrefSize(width, 800);

//System.out.println("going in tree area constructor");

this.btree = btree;

this.startX = startX;

this.startY = startY;

}

//setting up the lengths for drawing tree

public void setLengths(int treeHeight) {

this.lenX = this.lenX + 10\*treeHeight;

this.lenY = this.lenY + 5 ;

}

//setting up the paneSize

public void setSizeTreePane(int treeHeight) {

if (treeHeight == 0) {

this.setPrefSize(1500, 800);

} else {

double width = this.lenX \* treeHeight \* 2;

//double width = 1000 \* treeHeight \* 2;

double height = 500 \* treeHeight;

this.startX = width / 2;

this.setPrefSize(width, height);

}

}

//method for rendering a node

private void drawNode(int num, double posX, double posY, boolean isSearching) {

String s = Integer.toString(num);

Rectangle node = new Rectangle(posX, posY, nodeWidth, nodeWidth);

if (isSearching && num == searchNode) {

node.setFill(Color.web("#f57f7f"));

Timeline timeline = new Timeline();

timeline.setCycleCount(5);

timeline.setAutoReverse(true);

timeline.getKeyFrames().add((

new KeyFrame(Duration.millis(0),

new KeyValue(node.scaleXProperty(),1 ),

new KeyValue(node.scaleYProperty(), 1)

)));

timeline.getKeyFrames().add((

new KeyFrame(Duration.millis(500),

new KeyValue(node.scaleXProperty(),1.3 ),

new KeyValue(node.scaleYProperty(), 1.3)

)));

timeline.getKeyFrames().add((

new KeyFrame(Duration.millis(1000),

new KeyValue(node.scaleXProperty(),1 ),

new KeyValue(node.scaleYProperty(), 1)

)));

timeline.setDelay(Duration.seconds(0.2));

timeline.play();

} else {

node.setFill(Color.rgb(24, 71, 102));

}

node.setArcWidth(15);

node.setArcHeight(15);

Text nodeValue = new Text(posX + 13 - s.length(), posY + 20, s);

nodeValue.setStyle("-fx-font-weight: bold; -fx-text-fill: white; -fx-font-size: 14");

nodeValue.setFill(Color.WHITE);

nodeValue.setStrokeWidth(3);

this.getChildren().addAll(node, nodeValue);

}

//method for drawing a tree

private void drawTree(Node root, double posX, double posY, double lengthX, double lengthY, boolean isSearch) {

//if root is not null, hence tree exists

try {

if (root != null || root.getLeftElement() == null) {

//Left element always exist, so rendering it first

drawNode((Integer) root.getLeftElement(), posX, posY, isSearch);

//If right element exists, then render that

if (!(root.getRightElement() == null)) {

drawNode((Integer) root.getRightElement(), posX + nodeWidth, posY, isSearch);

}

//Going through all the children of the current root

//ArrayList<Node> children = new ArrayList<>();

HashMap<Integer, Node> children;

if (root.getRightElement() == null) {

//right exists

children = new HashMap<>();

if (root.getLeftNode() != null) {

System.out.println("Left Node Added");

children.put(-1, root.getLeftNode());

}

if (root.getMidNode() != null) {

System.out.println("Middle node added");

children.put(1, root.getMidNode());

}

// if (root.getRightNode() != null) {

// System.out.println("Right node added");

// children.put(1, root.getRightNode());

// }

} else {

children = new HashMap<>();

if (root.getLeftNode() != null) {

System.out.println("Left Node Added");

children.put(-1, root.getLeftNode());

}

if (root.getMidNode() != null) {

System.out.println("Middle node added");

children.put(0, root.getMidNode());

}

if (root.getRightNode() != null) {

System.out.println("Right node added");

children.put(1, root.getRightNode());

}

}

if (root.isLeaf()) {

//Its a leaf node, so no children, hence no more things need to be rendered

} else {

double updatedY = posY + lengthY;

for (int i : children.keySet()) {

System.out.println(i);

double updatedX = posX + i \* lengthX;

Line line = new Line(posX + nodeWidth/2, posY + nodeWidth+2, updatedX + nodeWidth/2, updatedY);

line.setStroke(Color.BLACK);

line.setStrokeWidth(3);

this.getChildren().add(line);

drawTree(children.get(i), updatedX, updatedY, lengthX / 3, lengthY, isSearch);

}

}

}

} catch (NullPointerException e) {

System.out.println("Null pointer error occurred");

}

}

//method for implementing search animation

public void searchTree(int nodeValue) {

try {

this.searchNode = nodeValue;

drawTree(btree.getRoot(), startX, startY, lenX, lenY, true);

} catch (Exception e) {

System.out.println("Error Occurred !");

}

}

//method for rendering tree

public void makeTree(Operations<Integer> newBTree) {

this.btree = new Operations<>();

this.btree = newBTree;

this.setLengths(this.btree.getHeight());

this.setSizeTreePane(this.btree.getHeight());

this.btree.inOrder();

drawTree(btree.getRoot(), startX, startY, lenX, lenY, false);

}

//method for resetting to new tree

public void resetTree() {

this.btree = new Operations<>();

this.btree.clear();

drawTree(btree.getRoot(), startX, startY, lenX, lenY, false);

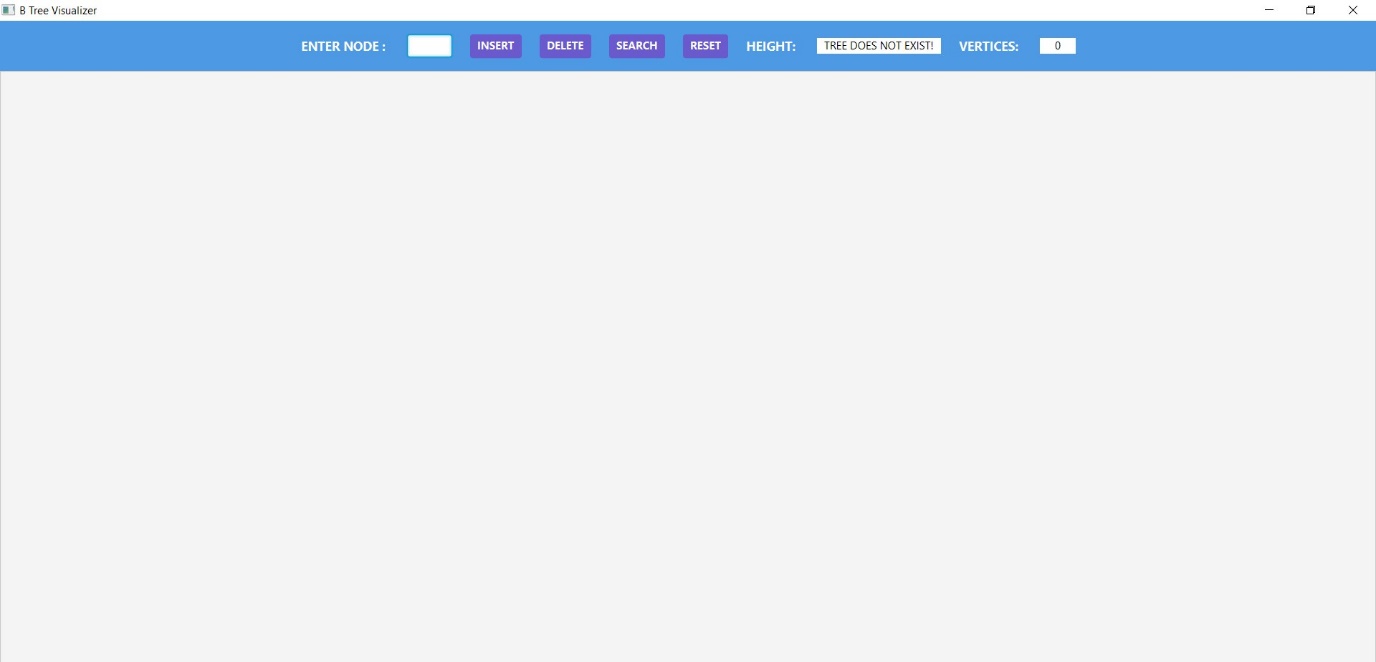
}

}

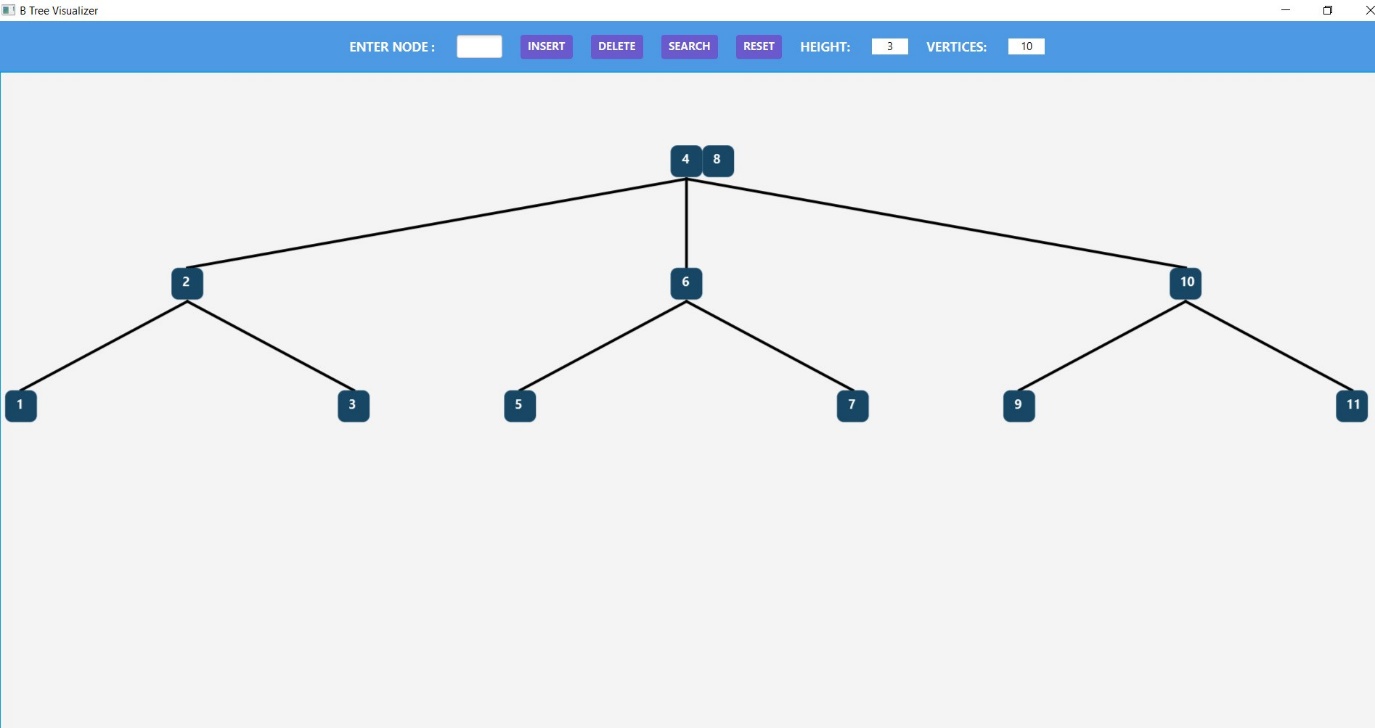
\*\*END OF CODE\*\*

SNAPSHOTS OF OUTPUT:

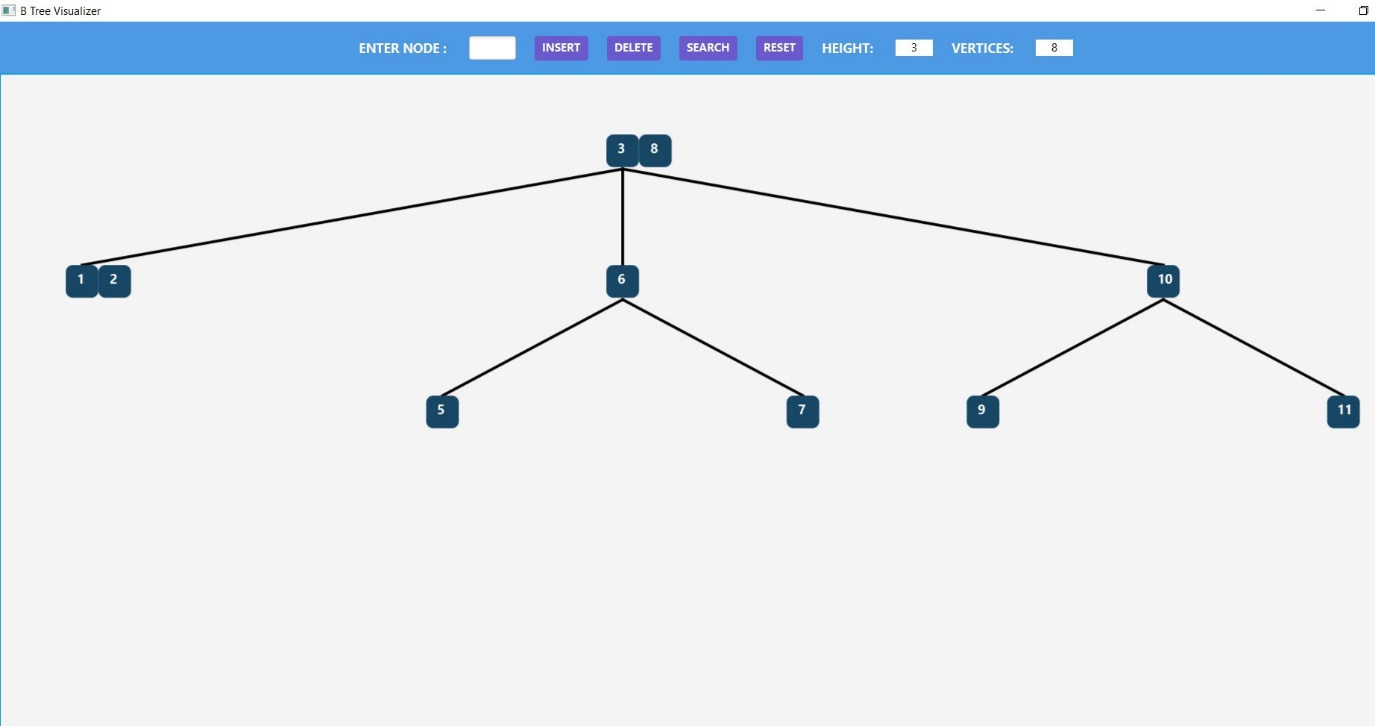
Initial window showing all the buttons, input field and the fields for height and vertices of our tree.



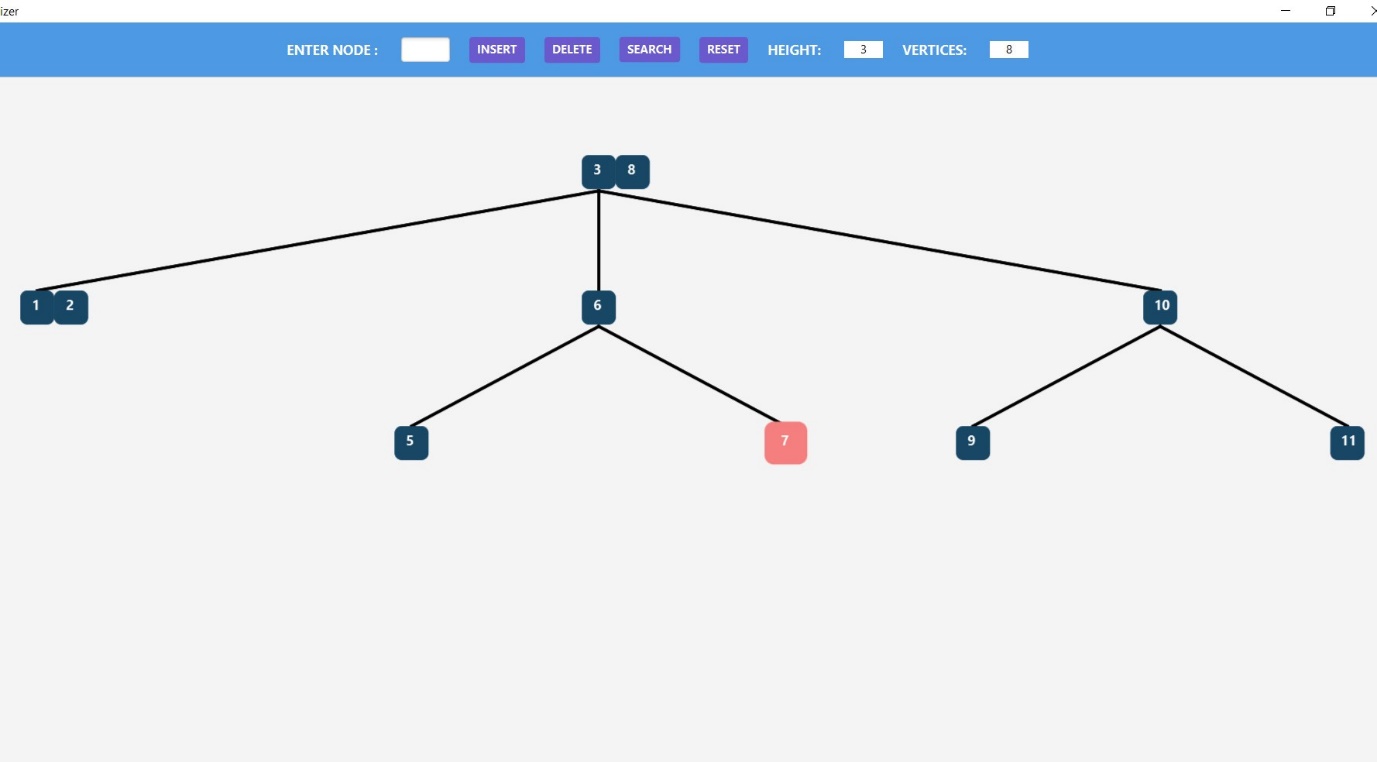
Inserting some nodes and we can see our tree is getting rendered, we can also see that the heights and vertices are changing accordingly as we are inserting new nodes.



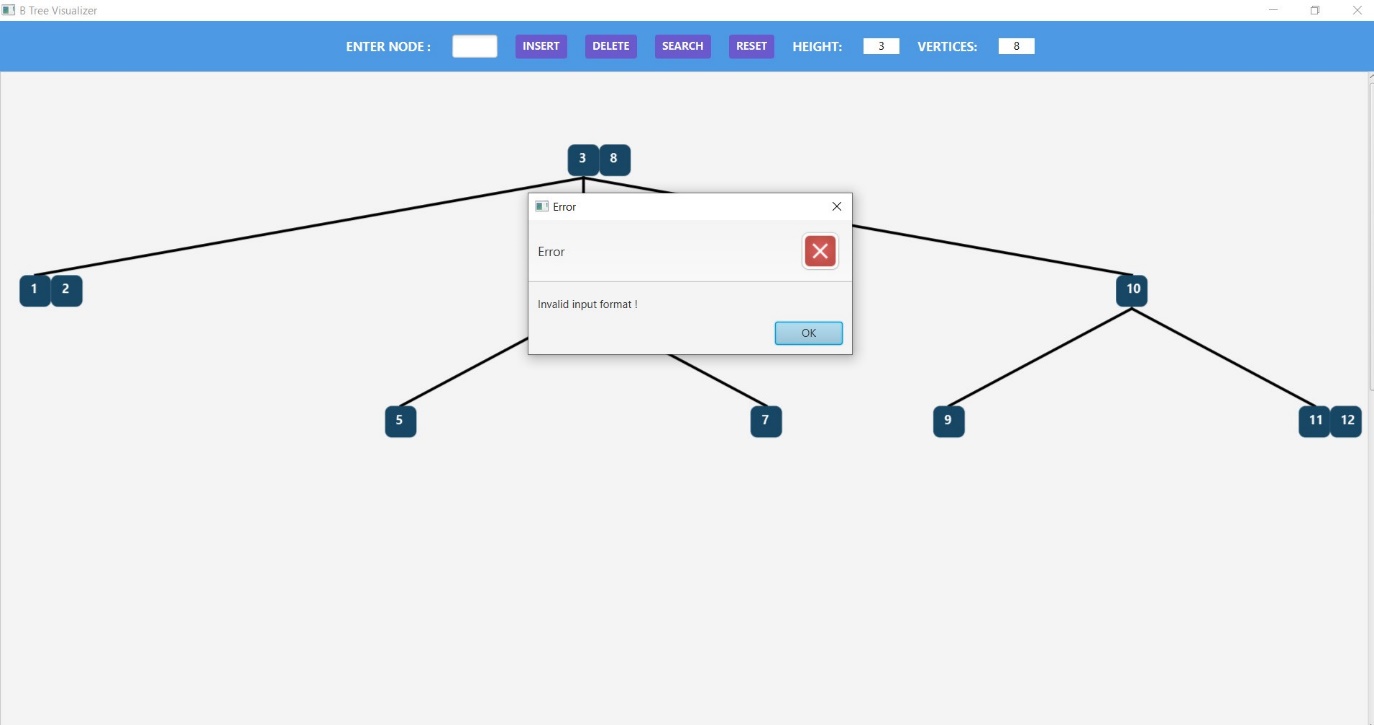
Deleting a node in our case we deleted 4.



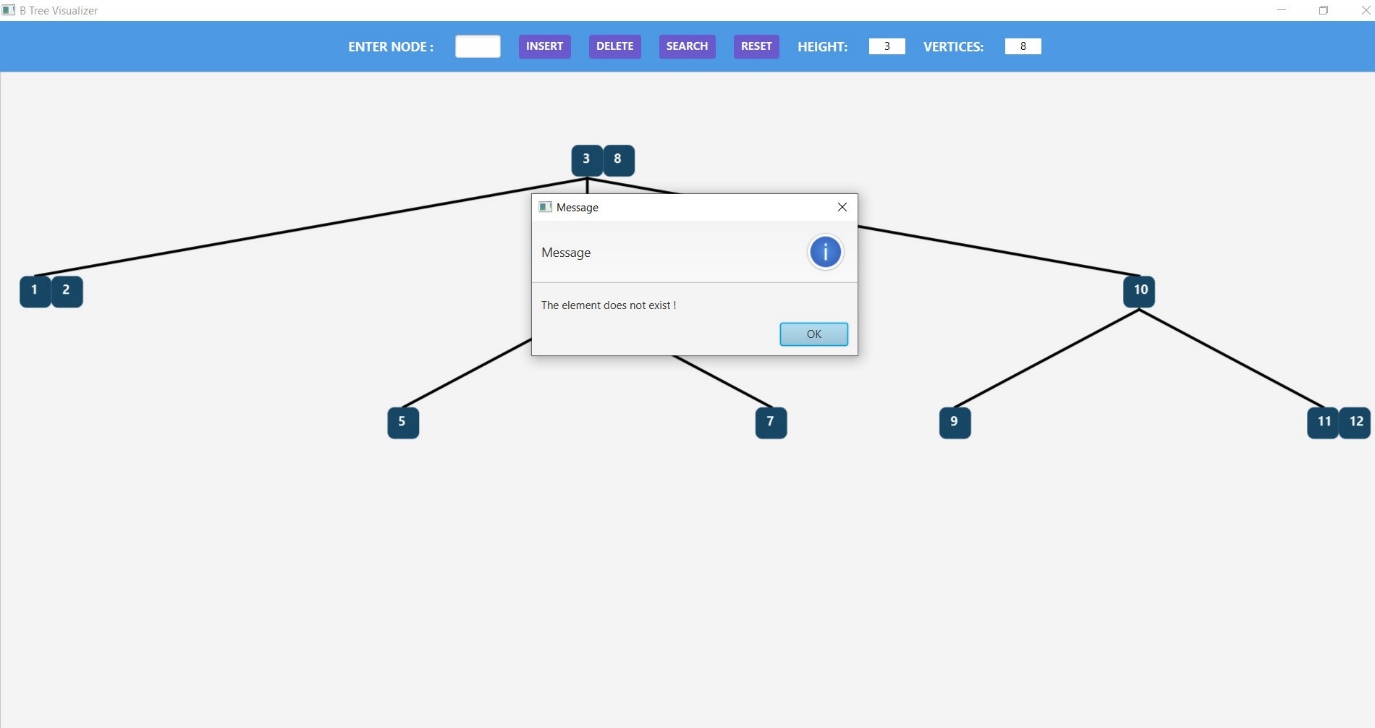
Searching an element say 7 in this case. Colour of the searched node changes and there is some animation.



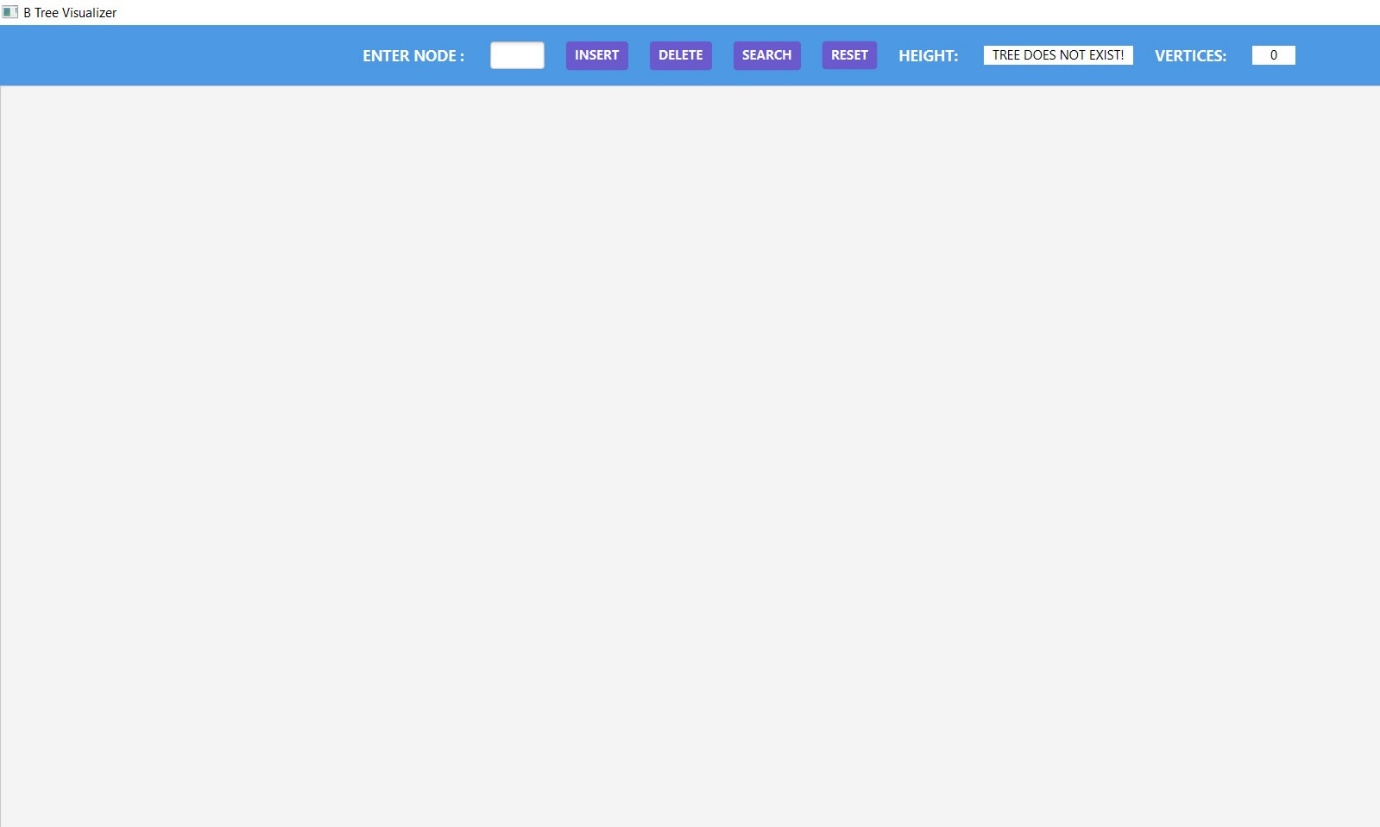
Displaying proper error messages for invalid input in case of insert/delete.



Displaying error message for an invalid search.



Resetting our tree. We can see height and vertices fields also gets reset.



\*\*END OF PROJECT\*\*

SHAGUN AGRAWAL: 1910110363

SIDDHANT MITTAL: 1910110388

BTECH CSE