HW-8

Question 1.

Consider the TreeNode class:

```
class TreeNode<E> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element){
        this.element = element;
    }

    public TreeNode(E element, TreeNode<E> left, TreeNode<E> right){
        this.element = element;
        this.left = left;
        this.right = right;
    }
}
```

Write each of the following pure functions:

- public static <E> ArrayList<E> inorder(TreeNode<E> root): This function returns the in-order traversal of the tree, whose root is referenced by root, as a list.
- public static <E> int depth(TreeNode<E> root): This function returns the depth of the tree, whose root is referenced by root. It is assumed that the depth of the root is 0.
- public static <E> ArrayList<E> level(TreeNode<E> root, int n): This function returns the values of the nodes on level n of the given tree as a list.
- public static <E> boolean symmetric(TreeNode<E> root): This function tests if the given binary tree is symmetric. A a binary tree is symmetric if rotating it about the vertical bar through the root for 180 degrees results in the same tree.
- public static Integer maxPathSum(TreeNode<Integer> root): This function returns the maximum path sum. A path sum of a leaf is the sum of the node values on the path from the root to the leaf.
- public static <E> boolean perfect(TreeNode<E> root): This function tests if the given binary tree is perfect.
- public static <E> boolean complete(TreeNode<E> root): This function tests if the given binary tree is complete.
- public static <E> boolean balanced(TreeNode<E> root): This function tests if the given binary tree is balanced.

- public static <E extends Comparable<E>> boolean bst(TreeNode<E> root): This function tests if the given binary tree is a binary search tree.
- public static <E extends Comparable<E>> boolean contains(TreeNode<E> root, E elm): This function tests if the element elm occurs in the given binary search tree.
- public static <E extends Comparable<E>> TreeNode<E> add(TreeNode<E> root, E elm): This function adds the value elm into the binary search tree, and returns the resulting tree, which must remain to be a binary search tree.

Question 2.

Considert the `BST` class given below. Write a class, named `MyBST`, that extends `BST` and provides the following methods:

- `public ArrayList<E> level(int n)`: This method returns the values of the nodes on level `n` of this tree as a list.
- `public boolean symmetric()`: This function tests if this binary tree is symmetric.
- `public boolean  perfect()`: This function tests if this binary tree is perfect.
- `public boolean  complete()`: This function tests if this binary tree is complete.
- `public boolean  balanced(TreeNode<E> root)`: This function tests if this binary tree is balanced.

```
public class BST<E extends Comparable<E>> implements Tree<E> {
  protected TreeNode<E> root;
  protected 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++)
      add(objects[i]);
  }

  @Override /** 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;
  }

  @Override /** Insert element e 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 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);
}
```

```java
@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);
}

/** This inner class is static, because it does not access
    any instance members defined in its outer class */
public static class TreeNode<E> {
  protected E element;
  protected TreeNode<E> left;
  protected TreeNode<E> right;

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

@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 java.util.ArrayList<TreeNode<E>> path(E e) {
  java.util.ArrayList<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; // Return an array list of nodes
}

@Override /** 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 at by current
  }

  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) {
```

```java
      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 inorder. */
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;
  }
}
```