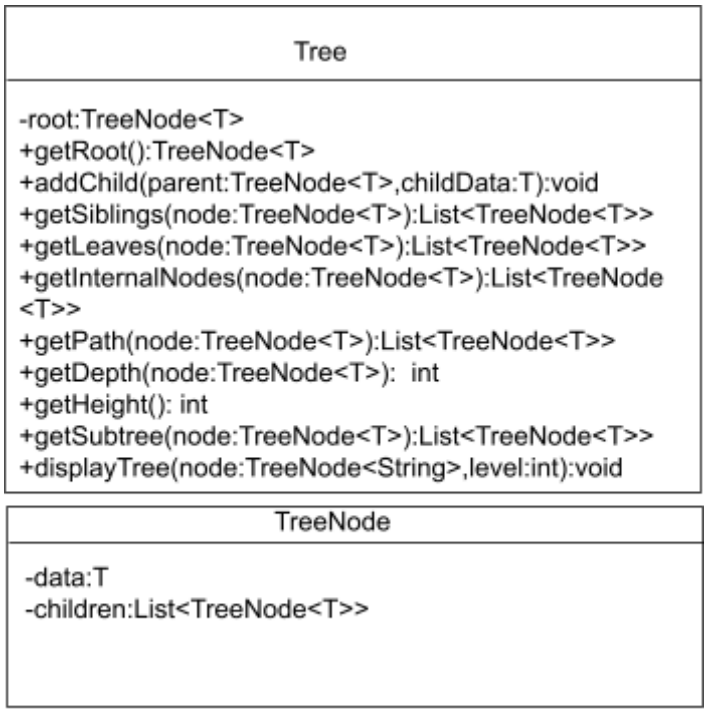
Laboratory-06

Experiment: To implement Tree data Structure by adding child node and displaying the tree data structure.Find the following properties of the tree/node:

1. Siblings of the node
2. List leaves of the tree
3. List internal nodes of the tree
4. List of edges
5. Path for a given node
6. Depth of the node
7. Height of the tree
8. Subtree rooted at given node

Class Diagram:



Java Constructs used-

• Importing Packages: It is essential to import packages like ArrayList and List to facilitate operations involving these data structures.

• Utilizing Tree Data Structure: Our implementation involves utilizing the ArrayList concept to create a dynamic array, forming the basis for our Tree Data Structure.

• Constructor Usage: Constructors, special method functions with the same name as the class, are automatically invoked upon object creation. In our experiment, the Tree constructor was utilized.

• Generic Class for Reusability: Employing a generic class enhances code reusability, allowing the same code to execute the data structure for various data types.

• Comparison Operators: The use of comparison operators like "==" and "!=" is prevalent to evaluate specific conditions within if-else statements.

• Return Keyword: The return keyword is used to yield either a null value or a specific value as declared when defining a function.

• Conditional Statements: Various conditional statements such as while loops, for loops (for tree traversal), if-else statements, and switch cases are employed to present a menu and execute operations based on user choices.

• Access Specifiers: Variables and methods are designated as public or private based on our requirements, controlling their accessibility.

Code

TreeNode.java

```
import java.util.*;
/**
 * Represents a node in a tree.
 *
 * @param <T> The type of data stored in the node. */
class TreeNode<T> {
 T data;
 List<TreeNode<T>> children;


 /**
 * Constructs a tree node with the specified data.  *
 * @param data The data for the node.  */
 public TreeNode(T data) {
 this.data = data;
 this.children = new ArrayList<>();
 }
}
```

Tree.java

```
import java.util.*;
/**
 * Represents a generic Tree data structure with various
operations.
 *
 * @param <T> The type of data stored in the tree nodes. */
class Tree<T> {
 private TreeNode<T> root;


 /**
 * Constructs a tree with the specified root data.  *
 * @param rootData The data for the root of the tree.  */
 public Tree(T rootData) {
 root = new TreeNode<>(rootData);
 }
 /**
 * Get the root of the tree.
 *
 * @return The root of the tree.
 */
 public TreeNode<T> getRoot() {
 return root;
 }
```

```
 /**
 * Adds a child node to the specified parent node.  *
 * @param parent The parent node.
 * @param childData The data for the new child node.  */
 public void addChild(TreeNode<T> parent, T childData) {
TreeNode<T> child = new TreeNode<>(childData);
parent.children.add(child);
 }

 /**
 * Finds siblings of the specified node.
 *
 * @param node The node to find siblings for.  *
@return List of siblings.
 */
 public List<TreeNode<T>> getSiblings(TreeNode<T> node) {  if (node ==
root) {
 return null; // Root has no siblings
 }

 TreeNode<T> parent = findParent(root, node);  if (parent
!= null) {
 return parent.children;
```

```
}

return null;
}

/**
* Finds leaves of the tree.
*
* @param node The node to start searching for leaves.
* @return List of leaf nodes.
*/
public List<TreeNode<T>> getLeaves(TreeNode<T> node) {
List<TreeNode<T>> leaves = new ArrayList<>();  findLeaves(node,
leaves);
return leaves;
}

/**
* Finds internal nodes of the tree.
*
* @param node The node to start searching for internal  nodes.
* @return List of internal nodes.
*/
```

```
 public List<TreeNode<T>> getInternalNodes(TreeNode<T>  node) {
 List<TreeNode<T>> internalNodes = new ArrayList<>();
findInternalNodes(node, internalNodes);
 return internalNodes;
 }

 /**
 * Finds the path from the root to the specified node.  *
 * @param node The node to find the path for.  *
@return List of nodes in the path.
 */
 public List<TreeNode<T>> getPath(TreeNode<T> node) {
List<TreeNode<T>> path = new ArrayList<>();  findPath(root, node,
path);
 return path;
 }

 /**
 * Finds the depth of the specified node in the tree.  *
 * @param node The node to find the depth for.  * @return
The depth of the node.
 */
 public int getDepth(TreeNode<T> node) {
 return findDepth(root, node, 0);
```

```
}

/**
* Find the height of the tree.
*
* @return The height of the tree.
*/
public int getHeight() {
return findHeight(root);
}

/**
* Finds the subtree rooted at the specified node.  *
* @param node The root of the subtree.  * @return The
subtree rooted at the specified node.  */
public Tree<T> getSubtree(TreeNode<T> node) {  TreeNode<T>
subtreeRoot = findNode(root, node);  if (subtreeRoot != null) {
Tree<T> subtree = new Tree<>(subtreeRoot.data);
copySubtree(subtreeRoot, subtree.getRoot());  return subtree;
}
return null;
}

// Other private methods...
```

```
 /**
 * Displays the tree starting from the specified node.  *
 * @param node The starting node for display.  *
@param level The current level in the tree.  */
 public void displayTree(TreeNode<String> node, int level) {
StringBuilder indent = new StringBuilder();  for (int i = 0; i < level; i++) {
 indent.append(" ");
 }
 System.out.println(indent.toString() + node.data);  for
(TreeNode<String> child : node.children) {  displayTree(child,
level + 1);
 }
 }


 /**
 * Finds a node with the specified data in the tree.  *
 * @param current The current node in the search.  * @param
targetData The data to search for.
 * @return The node with the specified data, or null if not  found.
 */
 public static TreeNode<String> findNode(TreeNode<String>  current,
String targetData) {
 if (current.data.equals(targetData)) {
```

```
 return current;
 }
 for (TreeNode<String> child : current.children) {  TreeNode<String>
found = findNode(child, targetData);  if (found != null) {
 return found;
 }
 }
 return null;
 }
}

/**
 * Represents a node in a tree.
 *
 * @param <T> The type of data stored in the node. */
class TreeNode<T> {
 T data;
 List<TreeNode<T>> children;

 /**
 * Constructs a tree node with the specified data.  *
 * @param data The data for the node.  */
 public TreeNode(T data) {
 this.data = data;
```

```
 this.children = new ArrayList<>();
 }
}
```

Main.java

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

/**
 * The Main class for interacting with the Tree data structure. */
public class Main {

 /**
 * Main method to execute the program.
 *
 * @param args Command-line arguments (not used in this  program).
 */
 public static void main(String[] args) {
 // Create a tree with the root node labeled "Root"
Tree<String> tree = new Tree<>("Root");
 Scanner scanner = new Scanner(System.in);
```

```
 // Display menu options
 System.out.println("1. Add child node\n" +  "2. Display
tree\n" +
 "3. Find siblings of a node\n" +
 "4. List leaves of the tree\n" +
 "5. List internal nodes of the tree\n" +  "6. List path for
a given node\n" +  "7. Depth of a node\n" +
 "8. Height of the tree\n" +
 "9. Subtree rooted at a given node\n" +  "10. Exit\n");


 boolean bool = true;
 while (bool) {
 System.out.print("Enter an operation:");  int choice =
scanner.nextInt();
 switch (choice) {
 case 1:
 // Add child node
 System.out.print("Enter parent node data: ");  String parentData
= scanner.next();  System.out.print("Enter child node data: ");
String childData = scanner.next();
 TreeNode<String> parentNode =
tree.findNode(tree.getRoot(), parentData);
 if (parentNode != null) {
```

```
 tree.addChild(parentNode, childData);
System.out.println("Child node added.");
 } else {
 System.out.println("Parent node not found.");  }
 break;
 case 2:
 // Display tree
 tree.displayTree(tree.getRoot(), 0);  break;
 case 3:
 // Find siblings of a node
 System.out.print("Enter node data: ");  String nodeData
= scanner.next();
 TreeNode<String> node =
tree.findNode(tree.getRoot(), nodeData);
 if (node != null) {
 List<TreeNode<String>> siblings =
tree.getSiblings(node);
 System.out.println("Siblings of the node:");  for
(TreeNode<String> sibling : siblings) {
System.out.println(sibling.data);  }
 } else {
 System.out.println("Node not found.");
 }
 break;
 case 4:
```

```
// List leaves of the tree
List<TreeNode<String>> leaves =
tree.getLeaves(tree.getRoot());
System.out.println("Leaves of the tree:");  for
(TreeNode<String> leaf : leaves) {
System.out.println(leaf.data);
}
break;
case 5:
// List internal nodes of the tree
List<TreeNode<String>> internalNodes =
tree.getInternalNodes(tree.getRoot());
System.out.println("Internal nodes of the tree:");
for (TreeNode<String> internalNode : internalNodes)  {
System.out.println(internalNode.data);  }
break;
case 6:
// List path for a given node
System.out.print("Enter node data: ");
nodeData = scanner.next();
node = tree.findNode(tree.getRoot(), nodeData);  if (node != null) {
List<TreeNode<String>> path = tree.getPath(node);
System.out.println("Path for the given node:");  for (TreeNode<String>
pathNode : path) {  System.out.println(pathNode.data);  }
```

```
} else {
System.out.println("Node not found.");  }
break;
case 7:
// Depth of a node
System.out.print("Enter node data: ");
nodeData = scanner.next();
node = tree.findNode(tree.getRoot(), nodeData);  if (node != null) {
int depth = tree.getDepth(node);
System.out.println("Depth of the node: " + depth);  } else {
System.out.println("Node not found.");  }
break;
case 8:
// Height of the tree
int height = tree.getHeight();
System.out.println("Height of the tree: " + height);  break;
case 9:
// Subtree rooted at a given node
System.out.print("Enter node data: ");  nodeData =
scanner.next();
node = tree.findNode(tree.getRoot(), nodeData);  if (node != null) {
Tree<String> subtree = tree.getSubtree(node);
System.out.println("Subtree rooted at the given  node:");
```

```
tree.displayTree(subtree.getRoot(), 0);  } else {
System.out.println("Node not found.");  }
break;
case 10:
// Exit
System.out.println("Exit.");
bool = false;
break;
default:
System.out.println("Invalid choice.");
}
}
}
}
```

Output:

```
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
1
Enter parent node data: Root
Enter child node data: Abhi
Child node added.
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
1
Enter parent node data: Root
Enter child node data: Aditya
Child node added.
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
2
Root
  Abhi
  Aditya
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
3
Enter node data: Abhi
Siblings of the node:
Abhi
Aditya
```

```
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
4
Leaves of the tree:
Abhi
Aditya
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
5
Internal nodes of the tree:
Root
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
6
Enter node data: Aditya
Path for the given node:
Root
Aditya
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
7
Enter node data: Abhi
Depth of the node: 1
```

```
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
8
Height of the tree: 1
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
9
Enter node data: Abhi
Subtree rooted at the given node:
Abhi
Choose an operation:
1. Add child node
2. Display tree
3. Find siblings of a node
4. List leaves of the tree
5. List internal nodes of the tree
6. List path for a given node
7. Depth of a node
8. Height of the tree
9. Subtree rooted at a given node
0. Exit
0
PS C:\Users\abhim\OneDrive\Desktop\Assignment6\Tree\src>
```

Errors encountered-

• Errors occurred due to neglecting to import the necessary modules in the appropriate places, preventing the implementation of functions reliant on those modules.
• Incorrectly declaring variables with incompatible data types resulted in errors, necessitating a review of variable requirements such as root, leaves, and nodes.
• Failure to provide expected inputs where necessary led to errors.
• Addressing errors involved ensuring the passage of accurate parameters with the correct data types according to the function's requirements.
• Encountering basic issues such as indentation and semicolon errors to some degree during the process

Learnings

I gained insights into the complexities involved in implementing a tree data structure, necessitating a deep understanding of its technical aspects, such as the distinct roles of the root, node, and leaf. Unlike binary trees, the flexibility of allowing any node to have multiple child nodes, each with its own set of children, adds a layer of complexity that requires careful analysis.
The diversity in implementation methods, utilizing various data structures, was apparent. In our experiment, we specifically delved into the intricacies of ArrayList, studying its functionalities comprehensively. I explored how to efficiently integrate it into a tree structure, highlighting the need for a thorough understanding of both the tree and the chosen data structure.

Conclusion

To conclude, implementing a Tree structure relies on adeptly utilizing an ArrayList and carefully handling root, leaf, and nodes. This approach ensures the seamless integration of data within the structure. Beyond this, exploring the Tree data structure unveils a spectrum of properties. These encompass critical attributes such as height,child nodes, parent nodes, leaves, depth, subtree, and more, adding depth to its understanding.. A comprehensive understanding of these properties is essential for navigating and harnessing the full potential of the Tree structure in various applications, offering a versatile and robust tool for organizing and processing hierarchical data.