JOBSHEET 14
Tree

14.1 Learning Objectives

After completing this practicum, students will be able to:

1. Understand tree models, particularly binary trees.
2. Design and declare the structure of binary tree algorithms.
3. Apply and implement binary tree algorithms in the context of binary search trees.

14.2 Experiment 1

Implementation of Binary Search Tree Using Linked List (100 Minutes)

14.2.1 Labs Activity

In this experiment, a Binary Search Tree will be implemented using a linked list, focusing on its fundamental operations.

1. In the project created during the previous session, we created a new package named **Week14**.
2. Create the following classes:
   a. Student00.java
   b. Node00.java
   c. BinaryTree00.java
   d. BinaryTreeMain00.java

   Change 00 with your order number.
3. Implement Class Student00, Node00, BinaryTree00 based on the following class diagram:

Student
nim: String name: String className: String ipk: double
Student() Student (nm: String, name: String, kls: String, ipk: double) print(): void

Node
data: Student left: Node right: Node
Node (left: Node, data:Student, right:Node)

BinaryTree
root: Node
BinaryTree() add(data: Student): void

find(ipk: double) : boolean traversePreOrder (node : Node) : void traversePostOrder (node : Node) void traverseInOrder (node : Node): void getSuccessor (del: Node) delete(ipk: double): void

4. In the **Student00** class, declare the attributes according to the **Student** class diagram shown above. Also, add the constructor and methods as specified in the diagram.

```
public class Student00 {
    String nim, name, className;
    double ipk;

    public Student00(){
    }
    public Student00(String nm, String nama, String kls, double ip){
        nim = nm;
        name = nama;
        className = kls;
        ipk = ip;
    }
    void print(){
        System.out.println(nim+" - "+name+" - "+className+" - "+ipk);
    }
}
```

5. In the **Node00** class, add the attributes **data**, **left**, and **right**, along with both a default constructor and a parameterized constructor.

```
public class Node00 {
    Student00 data;
    Node00 left;
    Node00 right;

    Node00(){
    }
    Node00(Student00 data){
        this.data = data;
        left = null;
        left = null;
    }
}
```

6. In the **BinaryTree00** class, add the attribute **root**.

```
public class BinaryTree00 {
    Node00 root;
}
```

7. Add a constructor and the **isEmpty()** method in the **BinaryTree00** class.

```
    public BinaryTree00(){
        root = null;
```

```
    }

    public boolean isEmpty(){
        return root == null;
    }
```

8. Add the **add()** method in the **BinaryTree00 class**. Nodes should be inserted into the binary search tree based on the student's GPA (IPK) value. In this case, the node insertion process is implemented non-recursively to make the insertion flow easier to understand. However, a recursive approach would result in more efficient and concise code.

```
    public void add(Student00 data){
        if(isEmpty()){
            root = new Node00(data);
        } else {
            Node00 current = root;
            while(true){
                if(data.ipk < current.data.ipk){
                    if(current.left != null){
                        current = current.left;
                    } else {
                        current.left = new Node00(data);
                        break;
                    }
                } else if(data.ipk > current.data.ipk){
                    if(current.right != null){
                        current = current.right;
                    } else {
                        current.right = new Node00(data);
                        break;
                    }
                } else {
                    break;
                }
            }
        }
    }
```

9. Add **find()** method

```
    public boolean find(double ipk){
        boolean result = false;
        Node00 current = root;
        while(current != null){
            if(current.data.ipk == ipk){
                result = true;
                break;
            } else if(ipk < current.data.ipk){
                current = current.left;
            } else {
                current = current.right;
            }
        }
        return result;
    }
```

10. Add the methods **traversePreOrder(), traverseInOrder()**, and **traversePostOrder()** to the **BinaryTree00** class. These traversal methods are used to visit and display the nodes in the binary tree using pre-order, in-order, and post-order traversal modes, respectively

```
public void traversePreOrder(Node00 node){
    if(node != null){
        node.data.print();
        traversePreOrder(node.left);
        traversePreOrder(node.right);
    }
}
public void traverseInOrder(Node00 node){
    if(node != null){
        traverseInOrder(node.left);
        node.data.print();
        traverseInOrder(node.right);
    }
}
public void traversePostOrder(Node00 node){
    if(node != null){
        traversePostOrder(node.left);
        traversePostOrder(node.right);
        node.data.print();
    }
}
```

11. Add the method **getSuccessor().** This method will be used during the deletion process of a node that has two children.

```
Node00 getSuccessor(Node00 del){
    Node00 successor = del.right;
    Node00 successorParent = del;
    while(successor.left != null){
        successorParent = successor;
        successor = successor.left;
    }
    if(successor != del.right){
        successorParent.left = successor.right;
        successor.right = del.right;
    }
    return successor;
}
```

12. Add the **delete()** method. Within this method, include a check to determine whether the tree is empty; if it is not, proceed to locate the node to be deleted.

```
public void delete(double ipk){
    if(isEmpty()){
        System.out.println("Tree is empty!");
        return;
    }
    Node00 parent = root;
    Node00 current = root;
    boolean isLeftChild = false;
    while(current.data.ipk != ipk){
        parent = current;
```

```
            if(ipk < current.data.ipk){
                isLeftChild = true;
                current = current.left;
            } else {
                isLeftChild = false;
                current = current.right;
            }
            if(current == null){
                System.out.println("Couldn't find data!");
                return;
            }
        }
        //delete node with no children
        if(current.left == null && current.right == null){
            if(current == root){
                root = null;
            } else if(isLeftChild){
                parent.left = null;
            } else {
                parent.right = null;
            }
        } else if(current.right == null){//delete node with a left child
            if(current == root){
                root = current.left;
            } else if(isLeftChild){
                parent.left = current.left;
            } else {
                parent.right = current.left;
            }
        } else if(current.left == null){//delete node with a right child
            if(current == root){
                root = current.right;
            } else if(isLeftChild){
                parent.left = current.right;
            } else {
                parent.right = current.right;
            }
        } else {//delete node with 2 children
            Node00 successor = getSuccessor(current);
            if(current == root){
                root = successor;
            } else if(isLeftChild){
                parent.left = successor;
            } else {
                parent.right = successor;
            }
            successor.left = current.left;
        }
    }
```

13. Open the **BinaryTreeMain00** class and add the **main**() method. Then, include the following code:

```
public class BinaryTreeMain00 {
    public static void main(String[] args) {
        BinaryTree00 bst = new BinaryTree00();
        bst.add(new Student00("244107020138", "Devin", "TI-1I", 3.57));
        bst.add(new Student00("244107020023", "Dewi", "TI-1I", 3.85));
```

```
        bst.add(new Student00("244107020225", "Wahyu", "TI-1I", 3.21));
        bst.add(new Student00("244107020076", "Angelina", "TI-1I", 3.54));

        System.out.println("Student list (in-order traversal)");
        bst.traverseInOrder(bst.root);

        System.out.println("Search data");
        System.out.print("Search a student with IPK: 3.54: ");
        String result = bst.find(3.54) ? "Found" : "Not Found";
        System.out.println(result);

        System.out.print("Search a student with IPK: 3.22: ");
        result = bst.find(3.22) ? "Found" : "Not Found";
        System.out.println(result);

        bst.add(new Student00("244107020223", "Andhika", "TI-1I", 3.72));
        bst.add(new Student00("244107020226", "Bima", "TI-1I", 3.37));
        bst.add(new Student00("244107020181", "Eiyu", "TI-1I", 3.46));
        System.out.println("Student list:");
        System.out.println("In-order traversal:");
        bst.traverseInOrder(bst.root);
        System.out.println("Pre-order traversal:");
        bst.traversePreOrder(bst.root);
        System.out.println("Post-order traversal:");
        bst.traversePostOrder(bst.root);

        System.out.println("Data deletion");
        bst.delete(3.57);
        System.out.println("Student list after deletion:");
        bst.traverseInOrder(bst.root);
    }
}
```

14. Compile and run the **BinaryTreeMain00** class to simulate the execution of the implemented binary tree program.

15. Observe and analyze the output generated during the program run.

```
Student list (in-order traversal)
244107020225 - Wahyu - TI-1I - 3.21
244107020076 - Angelina - TI-1I - 3.54
244107020138 - Devin - TI-1I - 3.57
244107020023 - Dewi - TI-1I - 3.85
Search data
Search a student with IPK: 3.54: Found
Search a student with IPK: 3.22: Not Found
Student list:
In-order traversal:
244107020225 - Wahyu - TI-1I - 3.21
244107020226 - Bima - TI-1I - 3.37
244107020181 - Eiyu - TI-1I - 3.46
244107020076 - Angelina - TI-1I - 3.54
244107020138 - Devin - TI-1I - 3.57
244107020223 - Andhika - TI-1I - 3.72
244107020023 - Dewi - TI-1I - 3.85
Pre-order traversal:
244107020138 - Devin - TI-1I - 3.57
244107020225 - Wahyu - TI-1I - 3.21
244107020076 - Angelina - TI-1I - 3.54
244107020226 - Bima - TI-1I - 3.37
244107020181 - Eiyu - TI-1I - 3.46
244107020023 - Dewi - TI-1I - 3.85
244107020223 - Andhika - TI-1I - 3.72
```

```
Post-order traversal:
244107020181 - Eiyu - TI-1I - 3.46
244107020226 - Bima - TI-1I - 3.37
244107020076 - Angelina - TI-1I - 3.54
244107020225 - Wahyu - TI-1I - 3.21
244107020223 - Andhika - TI-1I - 3.72
244107020023 - Dewi - TI-1I - 3.85
244107020138 - Devin - TI-1I - 3.57
Data deletion
Student list after deletion:
244107020225 - Wahyu - TI-1I - 3.21
244107020226 - Bima - TI-1I - 3.37
244107020181 - Eiyu - TI-1I - 3.46
244107020076 - Angelina - TI-1I - 3.54
244107020223 - Andhika - TI-1I - 3.72
244107020023 - Dewi - TI-1I - 3.85
```

14.2.2 Questions

1. Why is data search in a binary search tree more efficient compared to a regular binary tree?
2. What are the purposes of the **left** and **right** attributes in the **Node** class?
3. a. What is the function of the **root** attribute in the **BinaryTree** class?
   b. When a **BinaryTree** object is first created, what is the initial value of **root**?
4. When the tree is empty and a new node is to be added, what process takes place?
5. Consider the following line of code inside the **add**() method. Explain in detail the purpose of this line of code.

```
if(data.ipk < current.data.ipk){
     if(current.left != null){
         current = current.left;
     } else {
         current.left = new Node00(data);
         break;
     }
} else if(data.ipk > current.data.ipk){
     if(current.right != null){
         current = current.right;
     } else {
         current.right = new Node00(data);
         break;
     }
}
```

6. Explain the steps involved in the **delete**() method when removing a node that has two children. How does the **getSuccessor**() method assist in this process?

14.3 Experiment 2

Implementation of Binary Tree Using Array (45 Minutes)

14.3.1 Labs Activity

1. In this binary tree implementation experiment using an array, the tree data is stored directly in an array and initialized within the **main**() method. Subsequently, the **in-order** traversal process will be simulated.
2. Create the classes **BinaryTreeArray00** and **BinaryTreeArrayMain00**, replacing **00** with your attendance number.
3. In the **BinaryTreeArray00** class, create the attributes **data** and **idxLast**. Also, implement the methods **populateData**() and **traverseInOrder**().

```
public class BinaryTreeArray00 {
    Student00[] data;
    int idxLast;
    public BinaryTreeArray00(){
        data = new Student00[10];
        idxLast = -1;
    }
    void populateData(Student00[] data, int idxLast){
        this.data = data;
        this.idxLast = idxLast;
    }
    void traverseInOrder(int idxStart){
        if(idxStart <= idxLast){
            if(data[idxStart] != null){
                traverseInOrder(2 * idxStart + 1);
                data[idxStart].print();
                traverseInOrder(2 * idxStart + 2);
            }
        }
    }
}
```

4. Then, in the **BinaryTreeArrayMain00** class, create the **main**() method and insert the code shown in the following image inside the method.

```
public class BinaryTreeArrayMain00 {
    public static void main(String[] args) {
        BinaryTreeArray00 bta = new BinaryTreeArray00();
        Student00 m1 = new Student00("244107020138", "Devin", "TI-1I", 3.57);
        Student00 m2 = new Student00("244107020023", "Dewi", "TI-1I", 3.85);
        Student00 m3 = new Student00("244107020225", "Wahyu", "TI-1I", 3.21);
        Student00 m4 = new Student00("244107020076", "Angelina", "TI-1I", 3.54);
        Student00 m5 = new Student00("244107020223", "Andhika", "TI-1I", 3.72);
        Student00 m6 = new Student00("244107020226", "Bima", "TI-1I", 3.37);
        Student00 m7 = new Student00("244107020181", "Eiyu", "TI-1I", 3.46);

        Student00[] data = {m1, m2, m3, m4, m5, m6, m7};
        bta.populateData(data, data.length-1);
        System.out.println("In-order traversal:");
        bta.traverseInOrder(0);
    }
}
```

5. Run the **BinaryTreeArrayMain00** class and observe the output!

```
In-order traversal:
244107020076 - Angelina - TI-1I - 3.54
244107020023 - Dewi - TI-1I - 3.85
244107020223 - Andhika - TI-1I - 3.72
244107020138 - Devin - TI-1I - 3.57
244107020226 - Bima - TI-1I - 3.37
244107020225 - Wahyu - TI-1I - 3.21
244107020181 - Eiyu - TI-1I - 3.46
```

14.3.2 Questions

1. What is the purpose of the **data** and **idxLast** attributes in the **BinaryTreeArray** class?

2. What is the function of the **populateData**() method?
3. What is the purpose of the **traverseInOrder**() method?
4. If a binary tree node is stored at index **2** in the array, at which indices are its left and right children located, respectively?
5. What is the purpose of the statement int **idxLast = 6** in Experiment 2, step 4?
6. Why are the **indices 2 * idxStart + 1** and **2 * idxStart + 2** used in the recursive calls, and how do they relate to the structure of a binary tree represented as an array?

14.4 Assignments

Times Allocated: 150 minutes

1. Implement a recursive method **addRekursif**() in the **BinaryTree00** class to add nodes recursively.
2. Create methods **getMinIPK**() and **getMaxIPK**() in the **BinaryTree00** class to retrieve and display the student data with the lowest and highest GPA(IPK) values stored in the binary search tree.
3. Develop a method **displayStudentsWithIPKAbove** (**double threshold**) in the **BinaryTree00** class to display student data whose GPA(IPK) exceeds a specified threshold (e.g., above 3.50) within the binary search tree.
4. Modify the **BinaryTreeArray00** class by adding the following methods:
   - **add(Student data)** to insert data into the binary tree represented as an array.
   - **traversePreOrder()** to perform a pre-order traversal of the binary tree.