Data Structure and Algorithm Practicum
Binary Tree

Name
Muhammad Baihaqi Aulia Asy'ari

NIM
2241720145

Class
1I

Department
Information Technology

Study Program
D4 Informatics Engineering

1 COMPETENCIES

- Students are able to understand the Binary tree model
- Students are able to create and declare the structure of a Binary tree algorithm.
- Students are able to apply and implement the Binary search tree

2 PRACTICUM 1

In this practicum, we will try to implement Binary search tree with basic operations, using arrays (practicum 2) and linked lists (practicum 1).

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

BinaryTree
root: Node size: int
DoubleLinkedList() add(data: int): void find(data: int): boolean traversePreOrder (node : Node) : void traversePostOrder (node : Node) void traverseInOrder (node : Node): void getSuccessor (del: Node) add(item: int, index:int): void delete(data: int): void

1. Create `Node` , `BinaryTree` dan `BinaryTreeMain`, class
2. In the `Node` class, add **data, left, and right** attributes as well as default and constructors with parameters

```java
public class Node {
    int data;
    Node left;
    Node right;

    public Node() {
    }

    public Node(int data) {
        this.left = null;
        this.data = data;
        this.right = null;
    }
}
```

3. In the `BinaryTree` class, add **root** attribute.

```java
public class BinaryTree {
    Node root;
}
```

4. Add a default constructor and the **isEmpty()** method in the **BinaryTree** class

```java
public BinaryTree() {
    root = null;
}

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

5. Create **add()** method in the **BinaryTree class**. In the following image, the process of adding nodes is **not done recursively**, so that it is easier to see the process of adding nodes in a tree. Actually, if it's done with recursive approach, the writing code will be more efficient.

```java
void add(int data) {
    if (isEmpty()) {
        root = new Node(data);
    } else {
        Node current = root;
        while (true) {
            if (data < current.data) {
                if (current.left != null) {
                    current = current.left;
                } else {
                    current.left = new Node(data);
                    break;
                }
            } else if (data > current.data) {
                if (current.right != null) {
                    current = current.right;
                } else {
                    current.right = new Node(data);
                    break;
                }
            } else {
                break;
            }
        }
    }
}
```

6. Create **find() method**

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

7. Create the **traversePreOrder(), traverseInOrder() and traversePostOrder()** methods. All of hese traverse methods are used to visit and display nodes in the tree.

```
void traversePreOrder(Node node) {
    if (node != null) {
        System.out.print(" " + node.data);
```

```
            traversePreOrder(node.left);
            traversePreOrder(node.right);
        }
    }

    void traversePostOrder(Node node) {
        if (node != null) {
            traversePostOrder(node.left);
            traversePostOrder(node.right);
            System.out.print(" " + node.data);
        }
    }

    void traverseInOrder(Node node) {
        if (node != null) {
            traverseInOrder(node.left);
            System.out.print(" " + node.data);
            traverseInOrder(node.right);
        }
    }
```

8. Add the **getSuccessor()** method. This method will be used during the process of deleting a node that has 2 children.

```
Node getSuccessor(Node del) {
    Node successor = del.right;
    Node 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;
}
```

9. Create the **delete() method**.

```
void delete(int data) {

}
```

In the delete method, we need to add a further validation whether the tree is empty or not. If not find the position of the node to be deleted.

```
if (isEmpty()) {
    System.out.println("Tree is empty");
    return;
}

Node parent = root;
Node current = root;
boolean isLeftChild = false;
while (current != null) {
    if (current.data == data) {
        break;
    } else if (data < current.data) {
        parent = current;
        current = current.left;
        isLeftChild = true;
    } else if (data > current.data) {
        parent = current;
        current = current.right;
        isLeftChild = false;
    }
}
```

Then add the process of deleting the current node that has been found.

```
if (current == null) {
    System.out.println("Couldn't find data");
    return;
} else {
    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.left == null) {
        if (current == root) {
            root = current.right;
        } else {
            if (isLeftChild) {
                parent.left = current.right;
            } else {
                parent.right = current.right;
            }
        }
    } else if (current.right == null) {
        if (current == root) {
            root = current.left;
        } else {
            if (isLeftChild) {
                parent.left = current.left;
            } else {
                parent.right = current.left;
            }
        }
    } else {
        Node successor = getSuccessor(current);
        if (current == root) {
            root = successor;
        } else {
            if (isLeftChild) {
                parent.left = successor;
            } else {
                parent.right = successor;
            }
        }
        successor.left = current.left;
    }
}
```

10. Open the **BinaryTreeMain** class and add the **main()** method.

```
public class BinaryTreeMain {
    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();

        bt.add(6);
        bt.add(4);
        bt.add(8);
        bt.add(3);
        bt.add(5);
        bt.add(7);
        bt.add(9);
        bt.add(10);
        bt.add(15);

        bt.traversePreOrder(bt.root);
        System.out.println();
        bt.traverseInOrder(bt.root);
        System.out.println();
        bt.traversePostOrder(bt.root);
        System.out.println();
        System.out.println("Find " + bt.find(5));
        bt.delete(8);
        bt.traversePreOrder(bt.root);
        System.out.println();
    }
}
```

11. Compile and run the **BinaryTreeMain** class to get more understanding of how the program tree was created.

12. Observe the results.

3 PRACTICUM 2

1. In this experiment, the data tree is stored in an array and entered directly from the **main()** method, and then the traversal process is simulated in the order.

2. Create`BinaryTreeArray` and`BinaryTreeArrayMain` class

3. Create **data and idxLast** attributes in the **BinaryTreeArray** class. Also create **populateData () and traverseInOrder ()** methods.

```
public class BinaryTreeArray {
    int[] data;
    int idxLast;

    public BinaryTreeArray() {
        data = new int[10];
    }

    void populateData(int[] data, int idxLast) {
        this.data = data;
        this.idxLast = idxLast;
    }

    void traverseInOrder(int idxStart) {
        if (idxStart <= idxLast) {
            traverseInOrder(2*idxStart+1);
            System.out.print(data[idxStart]+" ");
            traverseInOrder(2*idxStart+2);
        }
    }
}
```

4. Then in the **BinaryTreeArrayMain** class create the **main()** method as shown below.

```
public class BinaryTreeArrayMain {
    public static void main(String[] args) {
        BinaryTreeArray bta = new BinaryTreeArray();
        int[] data = {6,4,8,3,5,7,9,0,0,0};
        int idxLast = 6;
        bta.populateData(data, idxLast);
        bta.traverseInOrder(0);
    }
}
```

5. Run the BinaryTreeArrayMain class and observe the results!

4 QUESTIONS

1. Why the data searching process is more efficient in the Binary search tree than in ordinary binary tree?
   `Answer:`
   because it use the devide and conque algorithm

2. Why do we need the **Node** class? what are the **left** and **right** attributes?
   `Answer:`
   the node class is used like in linked list, to link between data or to branch the data from node to node. left node and right node are used as a branching link from the current node.

3. a. What are the uses of the root attribute in the **BinaryTree** class?
   `Answer:`

the root is used like head in linked list to be the starting point of the data structure.

b. When the tree object was first created, what is the value of **root**?
`Answer:`
it started as null.

4. When the tree is still empty, and a new node is added, what process will happen?

   `Answer:`
   it will simply add data to the root node.

5. Pay attention to the **add()** method, in which there are program lines as below. Explain in detail what the program line is for?

```
if(data<current.data){
    if(current.left!=null){
        current = current.left;
    }else{
        current.left = new Node(data);
        break;
    }
}
```

   `Answer:`
   if the data is less than the current node data, then because the data is less than the current node data, it checks the current left node if its null or not. if the current left node is not null, then the current node will move to the current left node. but if not then it'll just add a new node to the current left node.

6. What is the difference between pre-order, in-order and post-order traverse modes?

   `Answer:`
   pre-order print the data before calling the recursive traversal. post order print the data after calling the recursive traversal. the in-order print the data in between calling the recursive traversal for left and right.

7. Look at the **delete()** method. Before the node removal process, it is preceded by the process of finding the node to be deleted. Besides intended to find the node to be deleted (current), the search process will also look for the parent of the node to be deleted (parent). In your opinion, why is it also necessary to know the parent of the node to be deleted?
   `Answer:`

because to remove the actuall node that we want to remove, we need to know where the pointer of the node is, which is located at it's parent node.

8. For what is a variable named isLeftChild created in the **delete()** method?
   `Answer:`
   it is used to move from node to node to find the node we want to delete and when we want to actually delete the data.

9. What is the **getSuccessor()** method for?
   `Answer:`
   to get the child node of the current node we're on that qualified to replaced the deleted data.

10. In a theoretical review, it is stated that when a node that has 2 children is deleted, the node is replaced by the successor node, where the successor node can be obtained in 2 ways, namely 1) looking for the largest value of the subtree to the left, or 2) looking for the smallest value of subtree on the right. Which 1 of 2 methods is implemented in the **getSuccessor()** method in the above program?
    `Answer:`
    it used the 2 method to look for the smallest value of subtree on the right

11. What are the uses of the data and idxLast attributes in the **BinaryTreeArray** class?
    `Answer:`
    to store the data and set the last index the program can use or to limit the program.

12. What are the uses of the **populateData()** and **traverseInOrder()** methods?

    `Answer:`
    it is used to insert data from an array and to print the data from the tree.

13. If a binary tree node is stored in index array 2, then in what index are the left-child and rigth child positions respectively?
    `Answer:`
    index 1 and 3.

5 ASSIGNMENTS

1. Create a method inside the **BinaryTree** class that will add nodes with recursive approach.

```java
void addRecursive(int data) {
    if (isEmpty()) {
        root = new Node(data);
    } else {
        Node current = root;
        addRecursiveNotNull(data, current);
    }
}

private void addRecursiveNotNull(int data, Node current) {
    if (data < current.data) {
        addTraverseLeftBranch(data, current);
    }
    if (data > current.data) {
        addTraverseRightBranch(data, current);
    }
}

private void addTraverseLeftBranch(int data, Node current) {
    if (current.left != null) {
        current = current.left;
        addRecursiveNotNull(data, current);
    } else {
        current.left = new Node(data);
    }
}

private void addTraverseRightBranch(int data, Node current) {
    if (current.right != null) {
        current = current.right;
        addRecursiveNotNull(data, current);
    } else {
        current.right = new Node(data);
    }
}
```

2. Create a method in the **BinaryTree** class to display the smallest and largest values in the tree.

```java
void printLargeAndSmall() {
    Node current = root;
    if (current.left == null && current.right == null) {
        System.out.printf("%s: %d \n", "Smallest and Largest",
         ↪  current.data);
    } else if (current.left == null) {
        System.out.printf("%s: %d \n", "Smallest", current.data);
        System.out.printf("%s: %d \n", "Largest",
         ↪  current.right.data);
    } else if (current.right == null) {
        System.out.printf("%s: %d \n", "Smallest",
         ↪  current.left.data);
        System.out.printf("%s: %d \n", "Largest", current.data);
    } else {
        printSearch(current.left, true);
        printSearch(current.right, false);
    }
}

private void printSearch(Node current, boolean isLeft) {
    if (isLeft && current.left != null) {
        current = current.left;
        printSearch(current, isLeft);
    } else if (!isLeft && current.right != null) {
        current = current.right;
        printSearch(current, isLeft);
    } else {
        System.out.printf("%s: %d \n", isLeft ? "Smallest" :
         ↪  "Largest", current.data);
    }
}
```

3. Create a method in the **BinaryTree** class to display the data in the leaf.

```
    void printLeaf() {
        Node current = root;
        if (isEmpty()) {
            System.out.println("is empty");
        } else {
            printLeafNotNull(current);
        }
    }

    private void printLeafNotNull(Node current) {
        if(current.left != null) printLeafNotNull(current.left);
        if(current.right != null) printLeafNotNull(current.right);
        if (current.left == null && current.right == null)
         ↪  System.out.print(current.data + ", ");
    }
```

4. Create a method in the **BinaryTree** class to display the number of leaves in the tree.

```
int countLeaf() {
    Node current = root;
    if (isEmpty()) {
        System.out.println("is empty");
    } else {
        return countLeafNotNull(current);
    }
    return 0;
}

private int countLeafNotNull(Node current) {
    int left = current.left != null ?
     ↪  countLeafNotNull(current.left) : 0;
    int right = current.right != null ?
     ↪  countLeafNotNull(current.right) : 0;
    int currentData = current.left == null && current.right ==
     ↪  null ? 1 : 0;
    return left + right + currentData;
}
```

5. Modify the **BinaryTreeMain** class, so that it has a menu option:

   a. add
   b. delete
   c. find
   d. traverse inOrder
   e. traverse preOrder
   f. traverse postOrder
   g. keluar

```
import java.util.Scanner;

public class BinaryTreeMain {
    static Scanner sc = new Scanner(System.in);
    static BinaryTree btTree = new BinaryTree();
    static void displayMenu() {
        System.out.println("======================");
        System.out.println("   Binary Tree Menu   ");
        System.out.println("======================");
        System.out.println("1. Add                ");
        System.out.println("2. Delete             ");
        System.out.println("3. Find               ");
        System.out.println("4. Traverse inOrder   ");
        System.out.println("5. Traverse preOrder  ");
        System.out.println("6. Traverse postOrder ");
        System.out.println("7. Exit               ");
    }

    static int getData() {
        System.out.print("data(int): ");
        return sc.nextInt();
    }

    static void add() {
        System.out.println("add data to tree");
        btTree.add(getData());
    }

    static void delete() {
        System.out.println("delete data in tree by data");
        btTree.delete(getData());
    }

    static void find() {
        System.out.println("find data in tree");
        btTree.find(getData());
    }

    static void traverseInOrder() {
        System.out.println("traverse the tree in-order method");
        btTree.traverseInOrder(btTree.root);
    }

    static void traversePreOrder() {
        System.out.println("traverse the tree in-order method");
        btTree.traversePreOrder(btTree.root);
    }

    static void traversePostOrder() {
        System.out.println("traverse the tree in-order method");
        btTree.traversePostOrder(btTree.root);
    }

    static void exit() {
        sc.close();
    }

    static void pivot() {
        displayMenu();
        int option = sc.nextInt();
        switch (option) {
            case 1 -> add();
            case 2 -> delete();
            case 3 -> find();
            case 4 -> traverseInOrder();
            case 5 -> traversePreOrder();
            case 6 -> traversePostOrder();
            case 7 -> exit();
        }
    }

    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();

        bt.add(6);
        bt.add(4);
        bt.add(8);
        bt.add(3);
        bt.add(5);
        bt.add(7);
        bt.add(9);
        bt.add(10);
        bt.add(15);

        bt.traversePreOrder(bt.root);
        System.out.println();
        bt.traverseInOrder(bt.root);
        System.out.println();
        bt.traversePostOrder(bt.root);
        System.out.println();
        System.out.println("Find " + bt.find(5));
        bt.delete(8);
        bt.traversePreOrder(bt.root);
        System.out.println();

        pivot();
        sc.close();
    }
}
```

6. Modify the **BinaryTreeArray** class, and add:

   a. Add add method (int data) to enter data into the tree

   ```
   void add(int data) {
       idxLast++;
       this.data[idxLast] = data;
   }
   ```

   b. **traversePreOrder()** and **traversePostOrder()** methods

   ```
   void traversePreOrder(int idxStart) {
       if (idxStart <= idxLast) {
           System.out.print(data[idxStart]+" ");
           traversePreOrder(2*idxStart+1);
           traversePreOrder(2*idxStart+2);
       }
   }

   void traversePostOrder(int idxStart) {
       if (idxStart <= idxLast) {
           traversePostOrder(2*idxStart+1);
           traversePostOrder(2*idxStart+2);
           System.out.print(data[idxStart]+" ");
       }
   }
   ```