JOBSHEET 14

Binary Tree

Name
Sherly Lutfi Azkiah Sulistyawati

NIM
2341720241

Class
1I

Major
Information Technology

Study Program
D4 Informatics Engineering

Practicum 1

Practice > Week14 > Node.java > ...

```java
package Week14;

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

    public Node() {

    }

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

Practice > Week14 > BinaryTree.java > ...

```java
package Week14;

public class BinaryTree {
    Node root;

    public BinaryTree() {
        root = null;
    }

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

    void add(int data) {
        if (isEmpty()) { //tree is empty
            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 { //data is already exist
                    break;
                }
            }
        }
    }

    boolean find(int data) {
        boolean hasil = false;
        Node current = root;
        while (current != null) {
```

```
            if (current.data == data) {
                hasil = true;
                break;
            } else if (data < current.data) {
                current = current.left;
            } else {
                current = current.right;
            }
        }
        return hasil;
    }

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

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

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

    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;
    }

    void delete(int data) {
        if (isEmpty()) {
            System.out.println(x:"Tree is empty!");
            return;
        }
        //find node (curent) that will be deleted
        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;
            }
            //deletion
            if (current == null) {
                System.out.println(x:"Couldn't find data!");
                return;
            } else {
                //if there is no child, simply delete it
                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 there is 1 child (right)
            if (current == root) {
                root = current.right;
            } else {
                if (isLeftChild) {
                    parent.left = current.right;
                } else {
                    parent.right = current.right;
                }
            }
        } else if (current.right == null) { //if there is 1 child (left)
            if (current == root) {
                root = current.left;
            } else {
                if (isLeftChild) {
                    parent.left = current.left;
                } else {
                    parent.right = current.left;
                }
            }
        } else { //if there are 2 childs
            Node successor = getSuccessor(current);
            if (current == root) {
                root = successor;
            } else {
                if (isLeftChild) {
                    parent.left = successor;
                } else {
                    parent.right = successor;
                }
            }
            successor.left = current.left;
        }
    }
  }
 }
}
```

```
package Week14;

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

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

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

```
6
4
3
5
8
7
9
10
15

3
4
5
6
7
8
9
10
15

3
5
4
7
15
10
9
8
6

Find true
6
4
3
5
9
7
10
15
```

Practicum 2

```java
package Week14;

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);
        }
    }
}
```

```java
package Week14;

public class BinaryTreeArrayMain {
    Run | Debug
    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(idxStart:0);
    }
}
```

```
3 4 5 6 7 8 9
PS D:\College\Semester 2\AlgoritmadanStrukturData>
```

Question

1. Why the data searching process is more efficient in the Binary search tree than in ordinary binary tree?
   - The data searching process is more efficient in a Binary Search Tree compared to an ordinary binary tree due to its sorted nature, allowing binary search.
2. Why do we need the **Node** class? what are the **left** and **right** attributes?
   - The Node class is essential for defining the structure of each element in the tree. The left and right attributes point to the child nodes.
3. a. What are the uses of the **root** attribute in the **BinaryTree** class?
   - The root attribute in the BinaryTree class is the starting point of the tree.

   b. When the tree object was first created, what is the value of **root**?
   - Initially, its value is null.
4. When the tree is still empty, and a new node is added, what process will happen?
   - When the tree is empty, the new node becomes the root.
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;
   }
}
```

   - The provided code snippet checks if the current node's data is greater than the new data. If the left child exists, it moves to the left child; otherwise, it adds the new node as the left child.
6. What is the difference between pre-order, in-order and post-order traverse modes?
   - Pre-order: Visit root, left subtree, right subtree.
   - In-order: Visit left subtree, root, right subtree.
   - Post-order: Visit left subtree, right subtree, root.
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?

- Knowing the parent of the node to be deleted is necessary to reassign pointers and maintain tree structure.

8. For what is a variable named isLeftChild created in the **delete()** method?
   - It indicates whether the node to be deleted is a left child, used during the deletion process to correctly reassign parent pointers.
9. What is the **getSuccessor()** method for?
   - It finds the successor node to replace a deleted node with two children, typically the smallest value in the right subtree.
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?
    - The getSuccessor() method implemented in the program uses the second approach: looking for the smallest value in the right subtree. This method ensures that the in-order sequence of the Binary Search Tree is maintained after deletion.
11. What are the uses of the data and idxLast attributes in the **BinaryTreeArray** class?
    - data: Stores the tree nodes.
    - idxLast: Tracks the last index of the node in the array.
12. What are the uses of the **populateData**() and **traverseInOrder**() methods?
    - populateData(): Fills the array with initial data.
    - traverseInOrder(): Traverses the array in in-order fashion.
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?
    - If a binary tree node is stored at index 2, the left-child is at index 22+1 = 5, and the right-child is at index 22+2 = 6.

Assignment

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

```
void addRecursive(int data) {
    root = addRecursive(root, data);
}

private Node addRecursive(Node current, int data) {
    if (current == null) {
        return new Node(data);
    }
    if (data < current.data) {
        current.left = addRecursive(current.left, data);
    } else if (data > current.data) {
        current.right = addRecursive(current.right, data);
    }
    return current;
}
```

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

```
int findSmallestValue() {
    return findSmallestValue(root);
}

int findSmallestValue(Node root) {
    return root.left == null ? root.data : findSmallestValue(root.left);
}

int findLargestValue() {
    return findLargestValue(root);
}

int findLargestValue(Node root) {
    return root.right == null ? root.data : findLargestValue(root.right);
}
```

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

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

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

```
int countLeaves(Node node) {
    if (node == null) {
        return 0;
    }
    if (node.left == null && node.right == null) {
        return 1;
    }
    return countLeaves(node.left) + countLeaves(node.right);
}
```

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

```
package Week14;

import java.util.Scanner;

public class BinaryTreeMain {
    Run | Debug
    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();
        Scanner sc = new Scanner(System.in);
        boolean exit = false;

        while (!exit) {
            System.out.println(x:"\nMenu:");
            System.out.println(x:"a. Add");
            System.out.println(x:"b. Delete");
            System.out.println(x:"c. Find");
            System.out.println(x:"d. Traverse InOrder");
            System.out.println(x:"e. Traverse PreOrder");
            System.out.println(x:"f. Traverse PostOrder");
            System.out.println(x:"g. Exit");
            System.out.print(s:"Choose an option: ");
            char option = sc.next().charAt(index:0);

            switch (option) {
                case 'a':
                    System.out.print(s:"Enter value to add: ");
                    int valueToAdd = sc.nextInt();
                    bt.add(valueToAdd);
                    break;
                case 'b':
                    System.out.print(s:"Enter value to delete: ");
                    int valueToDelete = sc.nextInt();
                    bt.delete(valueToDelete);
                    break;
```

```
                case 'c':
                    System.out.print(s:"Enter value to find: ");
                    int valueToFind = sc.nextInt();
                    System.out.println("Found: " + bt.find(valueToFind));
                    break;
                case 'd':
                    bt.traverseInOrder(bt.root);
                    break;
                case 'e':
                    bt.traversePreOrder(bt.root);
                    break;
                case 'f':
                    bt.traversePostOrder(bt.root);
                    break;
                case 'g':
                    exit = true;
                    break;
                default:
                    System.out.println(x:"Invalid option. Please try again.");
            }
        }
        sc.close();
    }
}

```

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

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

```
BinaryTreeArray(int size) {
    data = new int[size];
    idxLast = -1;
}

void add(int data) {
    if (idxLast < this.data.length - 1) {
        this.data[++idxLast] = data;
    } else {
        System.out.println(x:"Tree is full");
    }
}
```

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

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

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