Midterm Exam - Design & Analysis of Algorithms (G)

Lecturer: Misbakhul Munir Irfaubakti, S.Kom., M.Sc.

Muhammad Febriansyah
5025211164

1. Based on MyTree.java and MyTreeOps.java above, please create a function, namely isBST() which has a recursive function inside this function. It checks whether a tree, i.e., MyTree t, is BST (Binary Search Tree). Hint: it is allowed to use a supported function for isBST(). Please update the function isBST() in file DAA1.java

Source Code

```
public static boolean isBST(MyTree t) {

      return isBST(t, Integer.MIN_VALUE, Integer.MAX_VALUE, 1);

  }

private static boolean isBST(MyTree t, int lowerBound, int upperBound, int i) {

      if (t == null)

          return true;

      i += 1;

      if (t.getValue() < lowerBound || t.getValue() > upperBound)

          return false;



      return (isBST(t.getLeft(), lowerBound, t.getValue() - 1, i)

              && isBST(t.getRight(), t.getValue() + 1, upperBound, i));

  }
```

Explanation

This code checks if a binary tree is a binary search tree (BST) by calling a private method with appropriate arguments.The public method calls the private method with four arguments: the MyTree object t, Integer.MIN_VALUE, Integer.MAX_VALUE, and 1. The second and third arguments represent the lower and upper bounds for the values in

the tree, while the fourth argument is not used in the private method. The private method uses recursion to check if the values of the nodes are within given bounds. If all values are within the bounds, the tree is a BST.

2. Please create a recursive function, namely printDescending() which receives an input of a BST t (where t is MyTree and its values are an integer), that can print the values of t in descending order. This function has to be created without making a separate list of values from t. Please update the function printDescending() in file DAA1.java

Source Code

```
public static void printDescending(MyTree t) {

    if (t == null)

        return;

    else if (!t.getEmpty()) {

        printDescending(t.getRight());

        System.out.println(" " + t.getValue());

        printDescending(t.getLeft());

    }

}
```

Explanation

The method is used to print all the values in the left subtree in descending order. first checks if the input tree t is null, if it is, the method returns without doing anything. If it is not This will print all the values in the right subtree in descending order. After that, it prints the value of the current node by calling t.getValue() and then recursively calls itself on the left subtree of t by calling t.getLeft().

3. Please create an efficient recursive function, namely max() which receives an input of a BST t (where t is MyTree and its values are an integer), that can get the maximum value of the t's values. It is not allowed to traverse and compare all the nodes in the tree. However, you should traverse at most one path in the tree from the root. It means this function works in O(log n) time for BST. Hint: assume we are a node x in BST, then all the values from the tree's left branches of x always have less than or equal (≤) values compared to the value of node x. So, the maximum value won't be existing in the tree's left branches. Where is the maximum value? Please update the function max() in file DAA1.java

Source Code

```
public static int max(MyTree t) {

    if (t == null || t.getEmpty()) {

        return Integer.MIN_VALUE;

    }

    return Math.max(t.getValue(), max(t.getRight()));

}
```

Explanation

This code is a method that finds the maximum value in a binary search tree. It checks if the tree is null or empty and returns the minimum integer value if it is. Otherwise, it returns the maximum value between the current node and the maximum value in the right subtree, obtained by recursively calling the method on the right subtree.

4. Please create a recursive function, namely isHeightBalanced() which receives an input MyTree t, that can check whether t has a balanced height (AVL tree condition). Please update the function isHeightBalanced() in file DAA2.java.

Source Code

```
public static boolean isHeightBalanced(MyTree t) {

```

```
        if (t.getEmpty()) {

            return true;

        } else {

            int height_right = MyTreeOps.height(t.getRight());

            int height_left = MyTreeOps.height(t.getLeft());



            if (Math.abs(height_left - height_right) <= 1) {

                        return  isHeightBalanced(t.getLeft())  &&
isHeightBalanced(t.getRight());

            } else {

                return false;

            }

        }

    }
```

Explanation

This method checks if a binary tree is height-balanced. A binary tree is height-balanced if the difference between the heights of the left and right subtrees of every node is no more than 1. The method calculates the heights of the left and right subtrees and checks if their difference is less than or equal to 1. If it is, the method recursively checks if the left and right subtrees are also balanced. If both subtrees are balanced, the whole tree is balanced and the method returns true. Otherwise, it returns false.

5. AVL tree is a Height-Balanced (HB) tree. Please create a recursive function, namely insertHB() which receives the inputs of int n and MyTree t, that can insert n into t while it keeps preserving the AVL condition. Please update the function insertHB() in file DAA2.java.

Source Code

```
    public static MyTree insertHB(int n, MyTree t) {

        // Write your codes in here

        if (t.getEmpty()) {

            return new MyTree(n, new MyTree(), new MyTree());

        } else if (n < t.getValue()) {

                    return  rebalanceForLeft(new  MyTree(t.getValue(),
insertHB(n, t.getLeft()), t.getRight()));

        } else if (n > t.getValue()) {

                   return  rebalanceForRight(new  MyTree(t.getValue(),
t.getLeft(), insertHB(n, t.getRight())));

        } else {

            return t;

        }

    }
```

Explanation

The insertHB method inserts a value into a height-balanced binary search tree while maintaining its height-balanced property. It checks if the tree is empty and creates a new tree if it is. Otherwise, it inserts the value into the left or right subtree accordingly and calls rebalanceForLeft or rebalanceForRight methods to rebalance the tree if necessary. If the value is equal to the current node, it returns the input tree unchanged.

6. Function name: private static MyTree rebalanceForLeft(MyTree t). Please update this function in file DAA2.java.

Source Code

```
    private static MyTree rebalanceForLeft(MyTree t) {

                            if    (MyTreeOps.height(t.getLeft())    <=
(MyTreeOps.height(t.getRight()) + 1))

            return t;



        MyTree to_left = t.getLeft(), to_right = t.getRight();

           MyTree  to_left_left  =  to_left.getLeft(),  to_left_right  =
to_left.getRight();



                            if    (MyTreeOps.height(to_left_left)    >
MyTreeOps.height(to_right)) {

              return new MyTree(to_left.getValue(), to_left_left, new
MyTree(t.getValue(), to_left_right, to_right));
```

```
        }



        return new MyTree(to_left_right.getValue(),

                    new MyTree(to_left.getValue(), to_left.getLeft(),

to_left_right.getLeft()),

                  new MyTree(t.getValue(), to_left_right.getRight(),

t.getRight()));

    }
```

Explanation

The rebalanceForLeft method rebalances an unbalanced binary search tree by performing a right or left-right rotation. It checks if the tree is already balanced by comparing the heights of its left and right subtrees. If the tree is not balanced, it retrieves its left and right subtrees and checks if a single right rotation is enough to rebalance the tree. If it is, it performs a right rotation. Otherwise, it performs a left-right rotation.

7. Function name: private static MyTree rebalanceForRight(MyTree t). Please update this function in file DAA2.java

Source Code

```
private static MyTree rebalanceForRight(MyTree t) {

        // Write your codes in here

                        if    (MyTreeOps.height(t.getRight())    <=

(MyTreeOps.height(t.getLeft()) + 1))

            return t;
```

```
        MyTree to_left = t.getLeft(), to_right = t.getRight();
         MyTree to_right_left = to_right.getLeft(), to_right_right =
to_right.getRight();

                          if    (MyTreeOps.height(to_right_right)    >
MyTreeOps.height(to_left)) {
                       return  new  MyTree(to_right.getValue(),  new
MyTree(t.getValue(), to_left, to_right_left), to_right_right);
        }

                  return  new  MyTree(to_right_left.getValue(),  new
MyTree(t.getValue(), t.getLeft(), to_right_left.getLeft()),
                                    new  MyTree(to_right.getValue(),
to_right_left.getRight(), to_right.getRight()));
        // Write your codes in here
    }
```

Explanation

The rebalanceForRight method rebalances an unbalanced binary search tree by performing a left or right-left rotation. It checks if the tree is already balanced by

comparing the heights of its left and right subtrees. If the tree is not balanced, it retrieves its left and right subtrees and checks if a single left rotation is enough to rebalance the tree. If it is, it performs a left rotation. Otherwise, it performs a right-left rotation.

8. Please create a recursive function, namely deleteHB() which receives the inputs of MyTree t and int x, that can delete x from t while it keeps preserving the AVL condition. Please update the function deleteHB() in file DAA2.java

Source Code

```
    public static MyTree deleteHB(MyTree t, int x) {

        if (t.getEmpty()) {

            return t;

        } else {

            int value = t.getValue();

            MyTree to_left = t.getLeft();

            MyTree to_right = t.getRight();



            if (x > value) {

                    return rebalanceForLeft(new MyTree(t.getValue(),
to_left, deleteHB(to_right, x)));

            } else if (x < value) {

                    return rebalanceForRight(new MyTree(t.getValue(),
deleteHB(to_left, x), to_right));
```

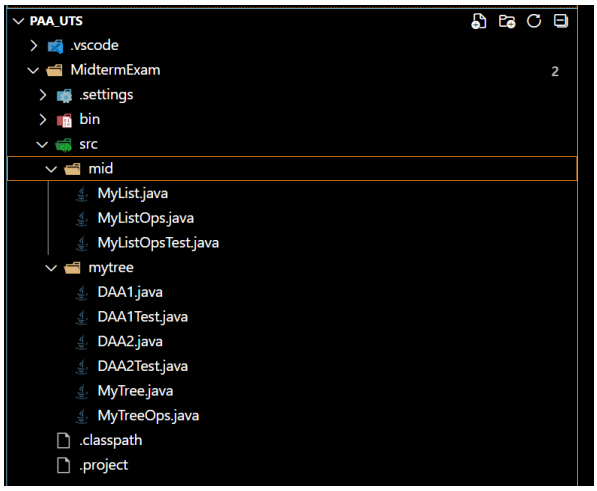
```
            } else {

                if (to_left.getEmpty()) {

                    return to_right;

                } else if (to_right.getEmpty()) {

                    return to_left;

                } else {

                    int greks = max(to_left);

                          return rebalanceForRight(new MyTree(greks,
deleteHB(to_left, greks), to_right));

                }

            }

        }

    }
```

Explanation

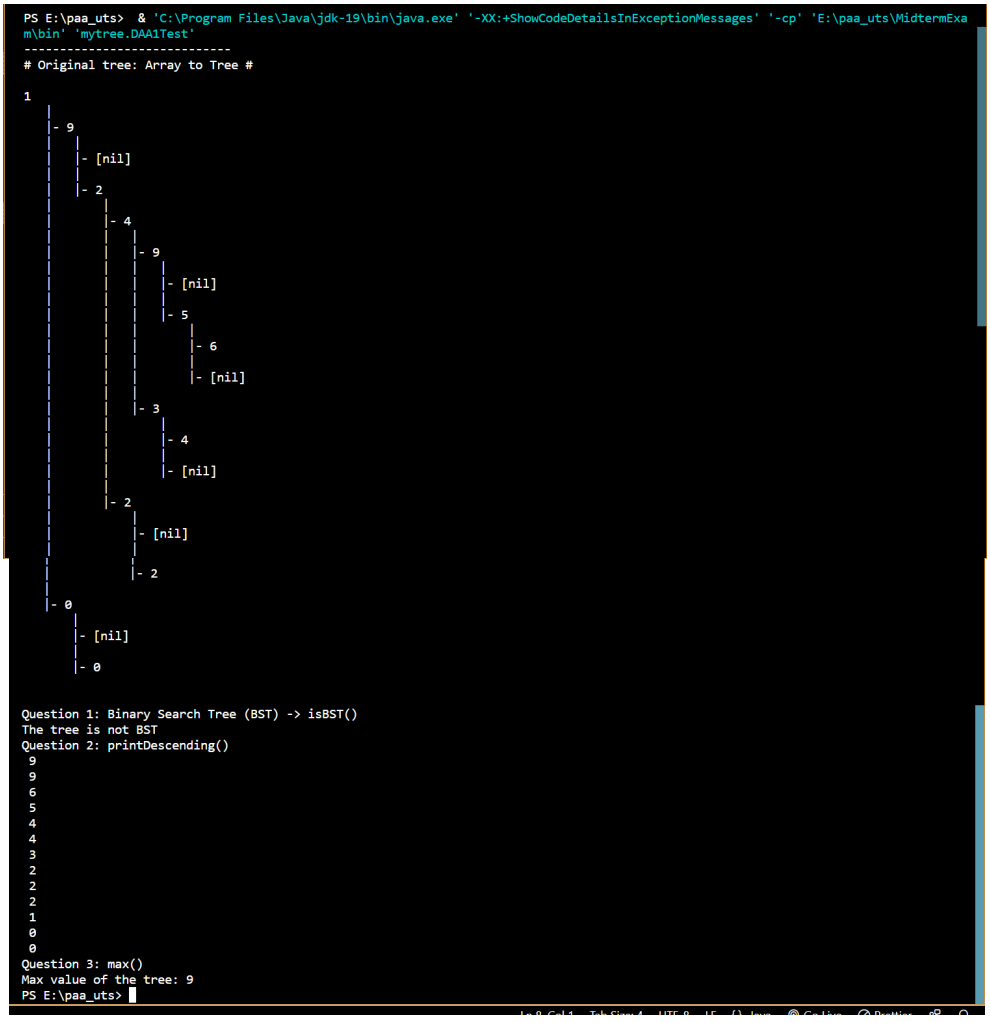
This method is used to delete a value from a height-balanced binary search tree while maintaining its height-balanced property. The deleteHB method takes in a MyTree object and an integer as arguments. It deletes the integer from the tree while maintaining its height-balanced property. The method uses recursion to traverse the tree and find the node to delete. If the node has no children or one child, it is replaced by its child. If it has two children, it is replaced by the maximum value in its left subtree. The method also calls rebalanceForLeft or rebalanceForRight methods to rebalance the tree if necessary.

OUTPUT EXECUTION

Working Template



DAA1Test.java



DAA2Test.java

```
PS E:\paa_uts>  & 'C:\Program Files\Java\jdk-19\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'E:\paa_uts\MidtermExam\bin' 'mytree.DAA2Test'
----------------------------
# Original tree: Array to Tree #

10
   |
   |- 16
   |   |
   |   |- [nil]
   |   |
   |   |- 15
   |
   |- 4
      |
      |- 5
      |
      |- [nil]


Question 4: isHeightBalanced()
The tree is Height-Balanced
Question 5: insertHB()
----------------------------
```

```
Question 4: isHeightBalanced()
The tree is Height-Balanced
Question 5: insertHB()
------------------------------
# 7 has been inserted #

10
    |
    |- 16
    |    |
    |    |- [nil]
    |    |
    |    |- 15
    |
    |- 4
         |
         |- 7
         |
         |- 5


The tree is Height-Balanced
------------------------------
```

```
----------------------------
# 12 has been inserted #

10
   |
   |- 4
   |   |
   |   |- 5
   |   |   |
   |   |   |- 12
   |   |   |
   |   |   |- [nil]
   |   |
   |   |- 7
   |
   |- 16
       |
       |- [nil]
       |
       |- 15


The tree is Height-Balanced
----------------------------
```

```
# 9 has been inserted #

10
   |
   |- 4
   |   |
   |   |- 5
   |   |   |
   |   |   |- 12
   |   |   |
   |   |   |- [nil]
   |   |
   |   |- 7
   |
   |- 15
       |
       |- 16
       |
       |- 9


The tree is Height-Balanced
```

```
Question 6: deleteHB()
----------------------------
# 7 has been deleted #

10
   |
   |- 4
   |   |
   |   |- 5
   |   |   |
   |   |   |- 12
   |   |   |
   |   |   |- [nil]
   |   |
   |   |- 7
   |
   |- 15
       |
       |- 16
       |
       |- 9


The tree is Height-Balanced
----------------------------
```

```
# 12 has been deleted #

10
   |
   |- 4
   |   |
   |   |- 5
   |   |
   |   |- 7
   |
   |- 15
       |
       |- 16
       |
       |- 9


The tree is Height-Balanced
----------------------------
# 9 has been deleted #

10
   |
   |- 4
   |   |
   |   |- 5
   |   |
   |   |- 7
   |
   |- 15
       |
       |- 16
----------------------------
# 10 has been deleted #

16
   |
   |- 4
   |   |
   |   |- 5
   |   |
   |   |- 7
   |
   |- 15


The tree is Height-Balanced
----------------------------
# 15 has been deleted #

4
   |
   |- 5
   |
   |- 16
       |
       |- 7
       |
       |- [nil]


The tree is Height-Balanced
PS E:\paa_uts>
```