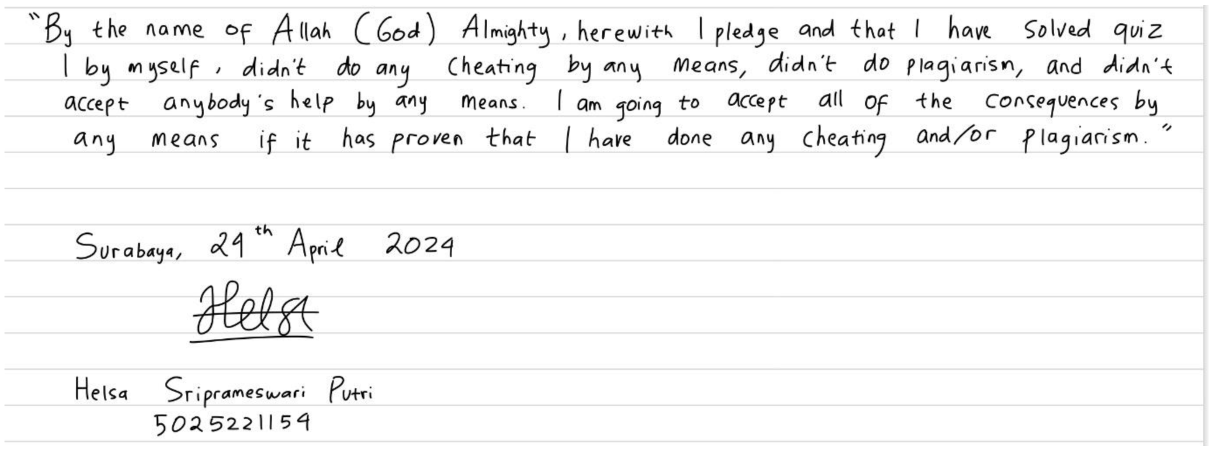
Midterm Exam Report

Full Name : Helsa Sriprameswari Putri

NRP : 5025221154

Class : DAA D



"By the name of Allah (God) Almighty, herewith I pledge and that I have solved quiz I by myself, didn't do any cheating by any means, didn't do plagiarism, and didn't accept anybody's help by any means. I am going to accept all of the consequences by any means if it has proven that I have done any cheating and/or plagiarism."

Surabaya, 29th April 2024

[illegible]

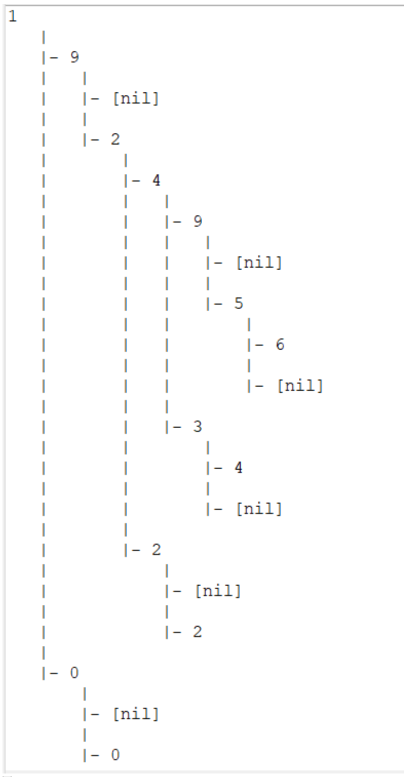
Helsa Sriprameswari Putri
5025221154

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.

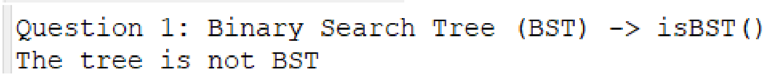
- Source Code

```java
public static boolean isBST(MyTree t) {
      return isBST(t, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    private static boolean isBST(MyTree t,  int lowerBound, int upperBound)
{
      if (t.getEmpty()) {
            return true;
        }

        int value = t.getValue();

        if (value > upperBound || value < lowerBound) {

            return false;
        }

        return isBST(t.getLeft(), lowerBound, value - 1) &&
isBST(t.getRight(), value + 1, upperBound);
    }
```

- Output



```
1
    |
    |- 9
    |    |
    |    |- [nil]
    |    |
    |    |- 2
    |         |
    |         |- 4
    |         |    |
    |         |    |- 9
    |         |    |    |
    |         |    |    |- [nil]
    |         |    |    |
    |         |    |    |- 5
    |         |    |         |
    |         |    |         |- 6
    |         |    |         |
    |         |    |         |- [nil]
    |         |    |
    |         |    |- 3
    |         |         |
    |         |         |- 4
    |         |         |
    |         |         |- [nil]
    |         |
    |         |- 2
    |              |
    |              |- [nil]
    |              |
    |              |- 2
    |
    |- 0
         |
         |- [nil]
         |
         |- 0
```



```
Question 1: Binary Search Tree (BST) -> isBST()
The tree is not BST
```

- Analysis :

`1. public static boolean isBST(MyTree t)`: This function check whether the provided binary tree t is a binary search tree. It will call the private helper method `isBST` with `t` and the minimum and maximum integer values.

`2.private static boolean isBST(MyTree t, int lowerBound, int upperBound)`: This method performs the recursive check for whether the binary tree t is a binary search tree within the range of `lowerBound` and `upperBound`.

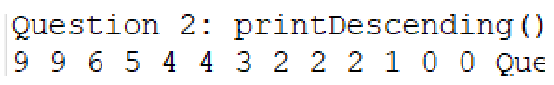
- **`t.getEmpty()`**: This method check if the current node of the tree is empty (i.e., null). If it is, it returns `true`, indicating that the subtree is a bst (as an empty subtree is a valid BST).
- **`int value = t.getValue()`**: This will get the value of the current node.
- **`if (value > upperBound || value < lowerBound)`**: It checks if the value of the current node falls outside the acceptable range defined by the `lowerBound` and `upperBound`. If it does, it means the tree violates the BST, and the function returns `false`.
- **`return isBST(t.getLeft(), lowerBound, value - 1) && isBST(t.getRight(), value + 1, upperBound)`**: This line checks whether the left and right subtrees of the current node are valid binary search trees. For the left subtree, the upper bound is set to `value - 1` (to make sure all values are less than the current node), and for the right subtree, the lower bound is set to `value + 1` (to make sure all values are greater than the current node). It returns `true` only if both left and right subtrees are BSTs; otherwise, it returns `false`.

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 must 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 && !t.getEmpty()) {

            printDescending(t.getRight());

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

            printDescending(t.getLeft());
         }
     }
```

- Output



```
Question 2: printDescending()
9 9 6 5 4 4 3 2 2 2 1 0 0 Que
```

- Analysis

The function `printDescending`, prints the values of a binary tree in descending order. It recursively traverses the tree, starting from the right subtree (since the bigger nodes are in the right subtree) , then prints the current node's value, and finally moves to the left subtree (to print the rest of smaller nodes).

- `if (t != null && !t.getEmpty()) {`: This checks if the tree `t` is not null and is not empty.
- `printDescending(t.getRight());`: This line recursively calls `printDescending` on the right subtree of `t` (since the bigger number is on the right of the subtree) , effectively traversing the tree in descending order.
- `System.out.print(t.getValue() + " ");`: This line prints the value of the current node `t` followed by a space.
- `printDescending(t.getLeft());`: This line recursively calls `printDescending` on the left subtree of `t`, completing the traversal of the tree in descending order.

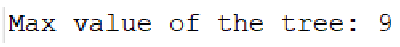
3. [10 points] Please create an efficient recursive function, namely max() which receives an input of a BST t (where t is MyTree and its values are an integer), that can get the maximum value of the t's values. It is not allowed to traverse and compare all the nodes in the tree. However, you should traverse 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 exist 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()) {
        throw new IllegalArgumentException("Empty BST!");
      }

      while (!t.getRight().getEmpty()) {
          t = t.getRight();
      }


      return t.getValue();
  }
```

- Output



```
Max value of the tree: 9
```

- Analysis

Since the maximum value of the tree is on the right side of the node, the code will work by visiting the right side of the tree and ignore the left side of the tree.

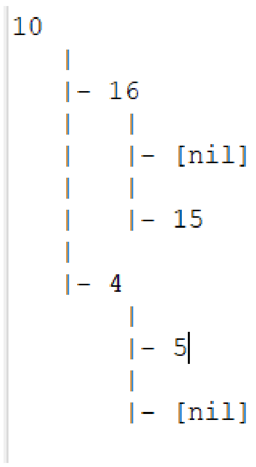
- `public static int max(MyTree t) {`: The method is to finds the maximum value in a binary search tree (`BST`).
- `if (t == null || t.getEmpty()) {`: This line checks if the input tree `t` is either `null` or empty. This ensures that the method does not attempt to find the maximum value in an empty tree.
- `while (!t.getRight().getEmpty()) {`: This line initiates a `while` loop that continues as long as the right child of the current node `t` is not empty. The loop traverses to the right child of `t` until it reaches the rightmost node of the tree, which will be the maximum value in a binary search tree.
- `t = t.getRight();`: Within the loop, it updates the variable `t` to point to its right child in each iteration. This effectively moves the traversal to the right subtree of the current node.
- `return t.getValue();`: Once the loop exits, the method returns the value stored in the rightmost node of the tree, which represents the maximum value in the binary search tree. This is because in a binary search tree, the rightmost node of any subtree will always contain the maximum value.

4. [10 points] 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.

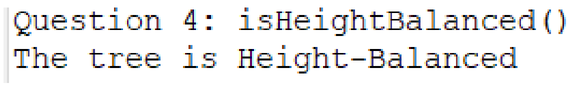
- Source Code

```
public static boolean isHeightBalanced(MyTree t) {
        if (t.getEmpty()) {
            return true;
        } else {
            int left_height = findHeight(t.getLeft());
            int right_height = findHeight(t.getRight());

            if (Math.abs(left_height - right_height) > 1) {
                return false;
            }

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

private static int findHeight(MyTree t) {
        if (t.getEmpty()) {
            return 0;
        } else {
            return 1 + Math.max(findHeight(t.getLeft()),
findHeight(t.getRight()));
        }
    }
```

- Output



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



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

- Analysis

  The `isHeightBalanced` method recursively checks if a binary tree is height-balanced by comparing the heights of its left and right subtrees. It utilizes the `findHeight` method to calculate the height of each subtree.

  1. isHeightBalanced Method:

     - The method takes a `MyTree` object `t` as its argument.
     - It first checks if the tree is empty (`t.getEmpty()`). If so, it returns `true` because an empty tree is considered height-balanced.

- If the tree is not empty, it calculates the height of the left and right subtrees using the `findHeight` method.
- It then checks if the absolute difference between the heights of the left and right subtrees is more than 1. If it is, the tree is not height-balanced, so it returns `false`.
- If the absolute difference is less than or equal to 1, it recursively checks whether both the left and right subtrees are height-balanced by calling `isHeightBalanced` on them.
- If both subtrees are height-balanced, it returns `true`; otherwise, it returns `false`.

2. findHeight Method:

- This is a helper method to calculate the height of a tree.
- It takes a `MyTree` object `t` as its argument.
- If the tree is empty, it returns 0 because an empty tree has a height of 0.
- If the tree is not empty, it recursively calculates the height of the left and right subtrees and returns the maximum of the two heights plus 1 (for the current node).

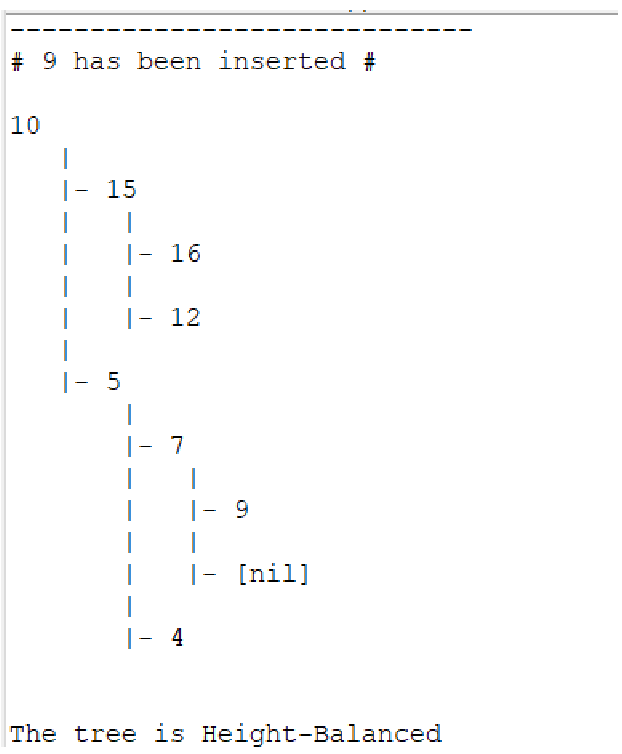
5. [10 points] The AVL tree is a Height-Balanced (HB) tree. Please create a recursive function, namely insertHB() which receives the inputs of int n and MyTree t, that can insert n into t while it keeps preserving the AVL condition. Please update the function insertHB() in file DAA2.java.

- Source Code

```
public static MyTree insertHB(int n, MyTree t) {
        if (t.getEmpty()) {
            return new MyTree(n, emptyTree, emptyTree);
        } else if (n < t.getValue()) {
            MyTree new_Left = insertHB(n, t.getLeft());
            return rebalanceForLeft(new_Left, t.getRight(), t.getValue());
        } else {
            MyTree new_Right = insertHB(n, t.getRight());
            return rebalanceForRight(t.getLeft(), new_Right, t.getValue());
        }
    }
```

- Output

```
# 7 has been inserted #

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


The tree is Height-Balanced
-----------------------------
# 12 has been inserted #

10
   |
   |- 15
   |   |
   |   |- 16
   |   |
   |   |- 12
   |
   |- 5
       |
       |- 7
       |
       |- 4


The tree is Height-Balanced
```



```
-----------------------------
# 9 has been inserted #

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


The tree is Height-Balanced
```

- Analysis

1. `insertHB` function:

- This function takes two parameters: `n` (the value to insert) and `t` (the current tree).
- It first checks if the current tree is empty. If it is, it creates a new tree node with value `n` and empty left and right subtrees, returning it.
- If the tree is not empty, it checks if the value `n` is less than the value of the current node (`t.getValue()`). If it is, it recursively inserts `n` into the left subtree (`t.getLeft()`) and then rebalances the tree for the left subtree using the `rebalanceForLeft` function.
- If `n` is greater than or equal to the value of the current node, it recursively inserts `n` into the right subtree (`t.getRight()`) and then rebalances the tree for the right subtree using the `rebalanceForRight` function.
- Finally, it returns the resulting balanced tree.

6. [15 points] Function name: private static MyTree rebalanceForLeft(MyTree t). Please update this function in file DAA2.java.

- Source Code

```
private static MyTree rebalanceForLeft(MyTree left, MyTree right, int
value) {
        if (findHeight(left) - findHeight(right) > 1) {
            if (findHeight(left.getLeft()) >= findHeight(left.getRight()))
{
                return rotateRight(left, right, value);
            } else {
                left = rotateLeft(left.getLeft(), left.getRight(),
left.getValue());
```

```java
                return rotateRight(left, right, value);
            }
        }
        return new MyTree(value, left, right);
    }

private static MyTree rotateRight(MyTree left, MyTree right, int value) {
        return new MyTree(left.getValue(), left.getLeft(), new
MyTree(value, left.getRight(), right));
    }

    private static MyTree rotateLeft(MyTree left, MyTree right, int value)
{
        return new MyTree(right.getValue(), new MyTree(value, left,
right.getLeft()), right.getRight());
    }
```

- Analysis

1. `rebalanceForLeft` function:
   a. This function takes three parameters: `left` (the left subtree), `right` (the right subtree), and `value` (the value being inserted).
   b. It checks if the height of the left subtree (`findHeight(left)`) minus the height of the right subtree (`findHeight(right)`) is greater than 1, indicating that the tree is unbalanced towards the left.
   c. If the left subtree is taller, it performs a <u>single rotation</u> to the right (`rotateRight`).
   d. If the right subtree of the left subtree is taller than its left subtree, it performs a <u>double rotation</u>: first a left rotation on the left child of `left`, then a right rotation on `left`.
   e. Finally, it returns the new balanced tree.
2. `rotateRight` function:
   a. This function performs a single right rotation on the given `left` subtree.
   b. It creates a new tree with the value of `left`, the left child of `left`, and a new subtree with the value passed to the function, the right child of `left`, and the `right` subtree.
3. `rotateLeft` function:
   a. This function performs a single left rotation on the given `right` subtree.
   b. It creates a new tree with the value of `right`, a new subtree with the value passed to the function, the `left` subtree, and the left child of `right`, and the right child of `right`

7. [15 points] Function name: private static MyTree rebalanceForRight(MyTree t). Please update this function in file DAA2.java.

- Source Code

```java
private static MyTree rebalanceForRight(MyTree left, MyTree right, int
value) {
        if (findHeight(right) - findHeight(left) > 1) {
            if (findHeight(right.getRight()) >=
findHeight(right.getLeft())) {
                return rotateLeft(left, right, value);
            } else {
                right = rotateRight(right.getRight(), right.getLeft(),
right.getValue());
                return rotateLeft(left, right, value);
            }
        }
        return new MyTree(value, left, right);
    }



private static MyTree rotateRight(MyTree left, MyTree right, int value) {
        return new MyTree(left.getValue(), left.getLeft(), new
MyTree(value, left.getRight(), right));
    }

    private static MyTree rotateLeft(MyTree left, MyTree right, int value)
{
        return new MyTree(right.getValue(), new MyTree(value, left,
right.getLeft()), right.getRight());
    }
```

- Analysis

1. `rebalanceForRight` function:

   a. This function also takes three parameters: `left` (the left subtree), `right` (the right subtree), and `value` (the value being inserted).
   b. It checks if the height of the right subtree (`findHeight(right)`) minus the height of the left subtree (`findHeight(left)`) is greater than 1, indicating that the tree is unbalanced towards the right.
   c. If the right subtree is taller, it performs a single rotation to the left (`rotateLeft`).
   d. If the left subtree of the right subtree is taller than its right subtree, it performs a double rotation: first a right rotation on the right child of `right`, then a left rotation on `right`.
   e. Finally, it returns the new balanced tree.

2. `rotateRight` function:

This function is the same as before that is mentioned above, performing a single right rotation on the given `left` subtree.

3. `rotateLeft` function:

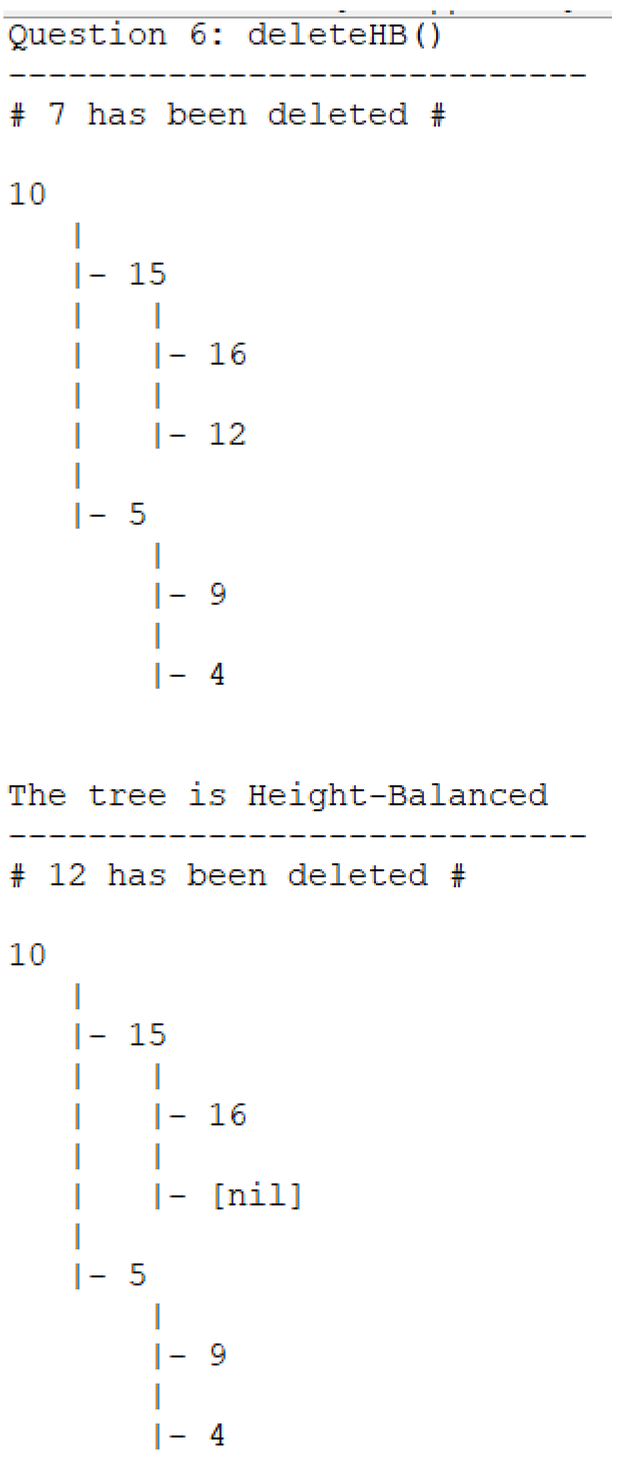
This function is also the same as before that is mentioned above, performing a single left rotation on the given `right` subtree.

8. [10 points] 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; // Value not found, return original tree
        }

        if (x < t.getValue()) {
            MyTree newLeft = deleteHB(t.getLeft(), x);
            return rebalanceForRight(newLeft, t.getRight(),
t.getValue());
        } else if (x > t.getValue()) {
            MyTree newRight = deleteHB(t.getRight(), x);
            return rebalanceForLeft(t.getLeft(), newRight,
t.getValue());
        } else {
            if (t.getLeft().getEmpty()) {
                return t.getRight();
            } else if (t.getRight().getEmpty()) {
                return t.getLeft();
            } else {
                int minRight = findMin(t.getRight());
                MyTree newRight = deleteHB(t.getRight(), minRight);
                return rebalanceForLeft(t.getLeft(), newRight,
minRight);
            }
        }
    }

private static int findMin(MyTree t) {
if (t.getLeft().getEmpty()) {
return t.getValue();
}
return findMin(t.getLeft());
}
```

- Output



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

10
   |
   |- 15
   |   |
   |   |- 16
   |   |
   |   |- 12
   |
   |- 5
       |
       |- 9
       |
       |- 4


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

10
   |
   |- 15
   |   |
   |   |- 16
   |   |
   |   |- [nil]
   |
   |- 5
       |
       |- 9
       |
       |- 4
```

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

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


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

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


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

```
-----------------------------
# 10 has been deleted #

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


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

5
   |
   |- 16
   |
   |- 4


The tree is Height-Balanced
```

- Analysis

1. deleteHB() Method:

   - This method is used to delete a node with value `x` from the tree `t`.
   - It first checks if the tree is empty. If it is, it simply returns the tree itself since there's nothing to delete.
   - Then it compares the value to be deleted (`x`) with the value of the current node.
     - If `x` is less than the current node's value, it recursively calls `deleteHB()` on the left subtree and then rebalances the tree if necessary.
     - If `x` is greater than the current node's value, it recursively calls `deleteHB()` on the right subtree and then rebalances the tree if necessary.
     - If `x` is equal to the current node's value, it handles three cases:

     1. If the node has no left child, it returns its right child.
     2. If the node has no right child, it returns its left child.
     3. If the node has both left and right children, it finds the minimum value in the right subtree (the leftmost node in the right subtree), replaces the current node's value with it, deletes the minimum node from the right subtree, and then rebalances the tree.

2. findMin() Method:

   This method finds the minimum value in a given binary search tree by recursively traversing left until it reaches a leaf node.

3. Rebalancing Methods:

   The rebalancing methods (`rebalanceForLeft()` and `rebalanceForRight()`)are responsible for maintaining the height balance property of the tree after deletion. These methods ensure that the tree remains height-balanced.