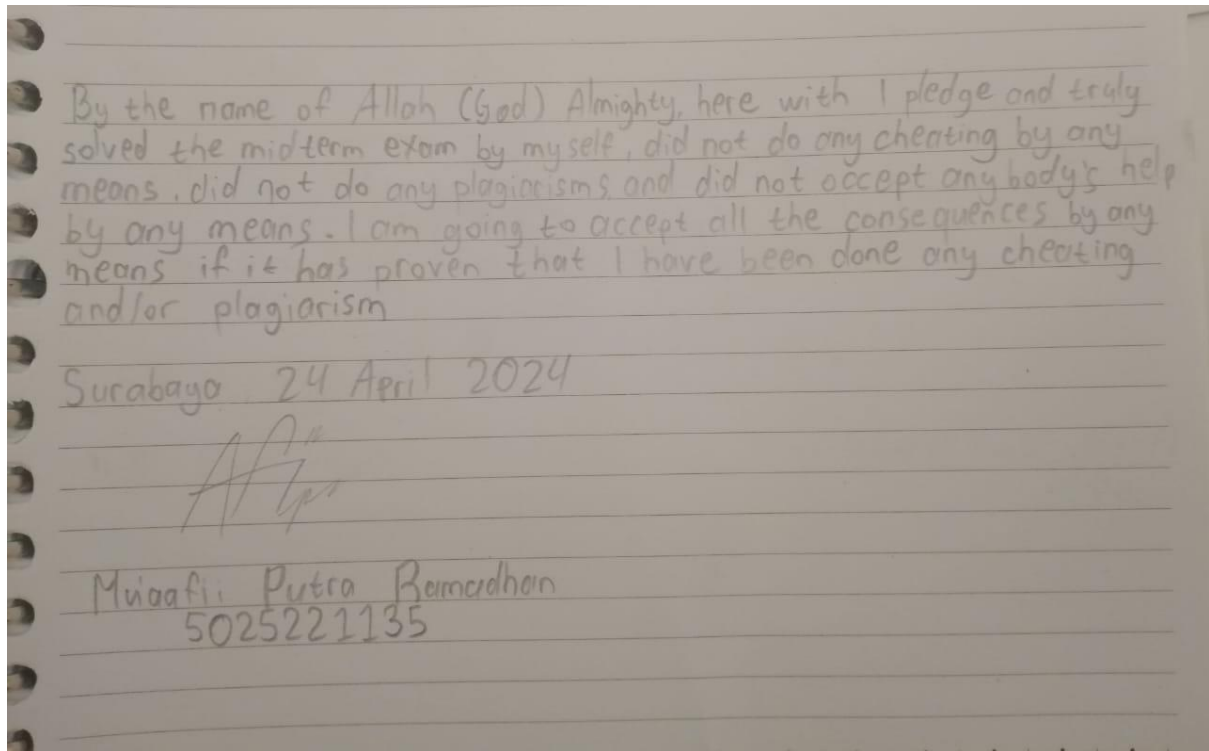


Name : Mu'aafii Putra Ramadhan
NRP : 50252211345
Class : Design & Analysis of Algorithm B

Midterm Test



Question

<https://drive.google.com/file/d/1I2Idw16G22qIEQ5dtVL155UQMaiypjv9/view>

Answer

DAA1.java

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.

```
// 1. isBST() [20 points]
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.isEmpty()) {
        return true;
    }
    int val = t.getValue();
    if (val <= lowerBound || val >= upperBound) {
        return false;
    }
    return isBST(t.getLeft(), lowerBound, val) && isBST(t.getRight(), val, upperBound);
}
```

This function takes a MyTree object t as input and returns a boolean (true if it's a BST, false otherwise). It delegates the actual checking to a private helper function.

Private Helper Function: isBST(MyTree t, int lowerBound, int upperBound)

This recursive function checks if the tree rooted at t follows the BST properties within the specified bounds (lowerBound and upperBound).

- Base Case: Empty Tree

If t is empty (using t.isEmpty()), it's considered valid as an empty BST. It returns true.

- Check Current Node Value

It retrieves the value of the current node using t.getValue().

If the value (val) is less than or equal to lowerBound or greater than or equal to upperBound, it violates the BST property and returns false. This ensures elements in the left subtree are always less than the root and right subtree elements are greater.

- Recursive Checks on Left and Right Subtrees

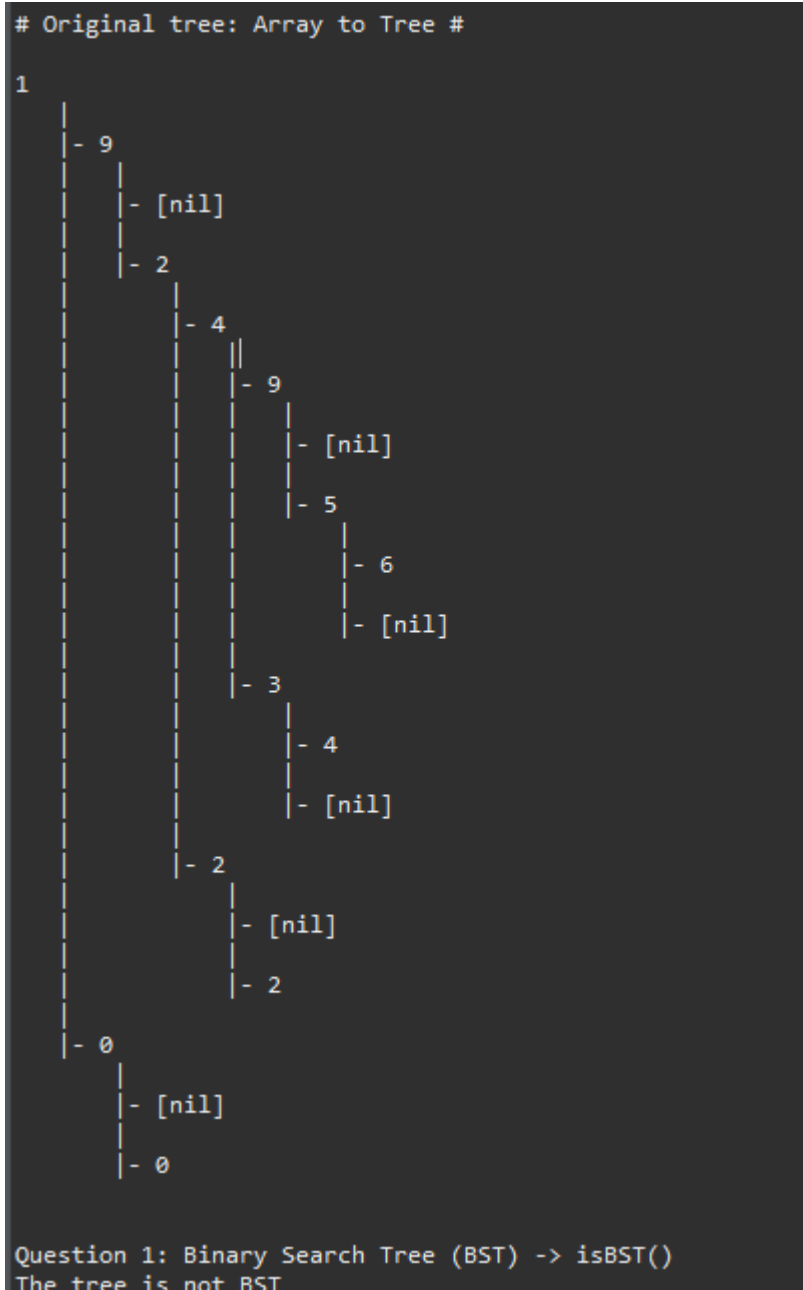
It recursively calls isBST on the left subtree (t.getLeft()) with the same lowerBound but an updated upperBound of the current node's value (val). This ensures elements in the left subtree remain within valid bounds.

Similarly, it calls isBST on the right subtree (t.getRight()) with an updated lowerBound of val (ensuring right subtree elements are greater) and the original upperBound.

- Combining Results

The function returns true only if both the left and right subtrees are valid BSTs (as checked by the recursive calls) and the current node's value itself conforms to the bounds.

Output :



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`.

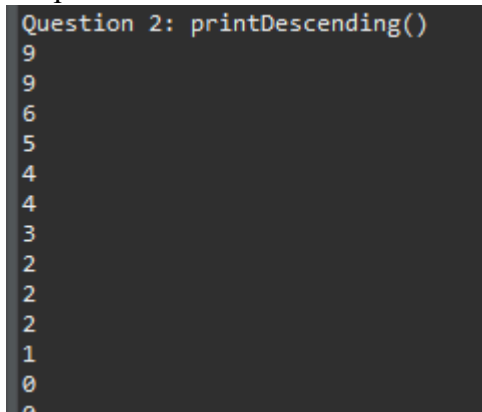
```
// 2. printDescending() [10 points]
public static void printDescending(MyTree t) {
    if (!t.isEmpty()) {
        printDescending(t.getRight());
        System.out.println(t.getValue());
        printDescending(t.getLeft());
    }
}
```

This function takes a `MyTree` object `t` as input and doesn't return anything (`void`). It uses in-order traversal with a twist to achieve descending order.

Recursive Traversal

- **Base Case: Empty Tree**
If `t` is empty (using `t.isEmpty()`), there's nothing to print. The function exits.
- **Right Subtree First**
It recursively calls `printDescending` on the right subtree `t.getRight()`. This is the key step for descending order. In a BST, the right subtree contains elements greater than the root.
- **Print Current Node**
After processing the right subtree (containing larger elements), it prints the value of the current node using `System.out.println(t.getValue())`.
- **Left Subtree Last**
Finally, it recursively calls `printDescending` on the left subtree `t.getLeft()`. Since the left subtree in a BST contains elements smaller than the root, processing it last ensures they are printed after the larger elements from the right subtree and the root itself.

Output :



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

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 (\leq) 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`.

```
// 3. max() [10 points]
public static int max(MyTree t) {
    if (t.isEmpty()) {
        throw new IllegalStateException("Attempted to find max in an empty tree");
    }

    MyTree current = t;
    while (!current.getRight().isEmpty()) {
        current = current.getRight();
    }
    return current.getValue();
}
}
```

First it checks if the tree `t` is empty using `t.isEmpty()`. If it is, it throws an `IllegalStateException` with a message indicating an attempt to find the max value in an empty tree. This prevents unexpected behavior and informs the user about the issue.

- **Iterative Approach**
It initializes a variable `current` to point to the root node (`t`).
It uses a while loop to traverse the tree. The loop continues as long as the right child of the current node (`current.getRight().isEmpty()`) is not empty.
- **Right Subtree Exploration**
In each iteration, the code assigns the right child of the current node to `current`. This effectively moves the pointer to the right subtree in the tree.
- **Reaching the Maximum**
The loop terminates when the right child of the current node becomes empty (`current.getRight().isEmpty()` is true). This indicates that the current node (`current`) is the rightmost node and therefore has the maximum value in the tree.
- **Returning the Maximum Value**
After the loop, the function returns the value of the current node (`current.getValue()`), which is the maximum value found in the tree.

Output :

```
Question 3: max()
Max value of the tree: 9
```

DAA2.java

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`.

```
// 4. isHeightBalanced() [10 points]
public static boolean isHeightBalanced(MyTree t) {
    return checkBalance(t).isBalanced;
}
// Helper class
private static class BalanceStatusWithHeight {
    boolean isBalanced;
    int height;

    BalanceStatusWithHeight(boolean isBalanced, int height) {
        this.isBalanced = isBalanced;
        this.height = height;
    }
}
// Recursive function to check if the tree is balanced
private static BalanceStatusWithHeight checkBalance(MyTree t) {
    if (t.isEmpty()) {
        return new BalanceStatusWithHeight(true, -1);
    }

    BalanceStatusWithHeight leftResult = checkBalance(t.getLeft());
    if (!leftResult.isBalanced) {
        return leftResult;
    }

    BalanceStatusWithHeight rightResult = checkBalance(t.getRight());
    if (!rightResult.isBalanced) {
        return rightResult;
    }

    boolean isBalanced = Math.abs(leftResult.height - rightResult.height) <= 1;
    int height = Math.max(leftResult.height, rightResult.height) + 1;
    return new BalanceStatusWithHeight(isBalanced, height);
}
```

This function takes a MyTree object t as input and returns a boolean (true if the tree is height balanced, false otherwise). It delegates the actual height balancing check to a helper function checkBalance.

- **Helper Function: checkBalance(MyTree t)**
This recursive function does the core work of checking if the tree rooted at t is height balanced. It returns a helper class BalanceStatusWithHeight that encapsulates both the balance status (isBalanced) and the height (height) of the subtree rooted at t.
- **Base Case: Empty Tree**
If the tree t is empty (t.isEmpty()), it's considered height balanced by definition. An empty tree has a height of -1 (as specified in the BalanceStatusWithHeight constructor) and is balanced. The function returns a BalanceStatusWithHeight object with isBalanced set to true and height set to -1.
- **Recursive Checks on Left and Right Subtrees**
The function calls checkBalance recursively on the left subtree (t.getLeft()) and stores the result in a BalanceStatusWithHeight variable named leftResult. Similarly, it calls checkBalance on the right subtree (t.getRight()) and stores the result in a BalanceStatusWithHeight variable named rightResult.

- Early Termination for Imbalanced Subtrees

If the left subtree (leftResult) is not balanced (i.e., leftResult.isBalanced is false), the function immediately returns leftResult. There's no need to continue checking the right subtree because an unbalanced subtree automatically makes the entire tree unbalanced.

The same logic applies to the right subtree (rightResult). If it's not balanced, the function returns rightResult.

- Checking Balance and Height of Current Subtree

If both the left and right subtrees are balanced (as checked in the previous steps), the function proceeds to calculate the balance status and height for the current subtree rooted at t.

It calculates the absolute difference in heights between the left and right subtrees using `Math.abs(leftResult.height - rightResult.height)`. A height balanced tree will have a maximum height difference of 1 between its subtrees.

It checks if the absolute height difference is less than or equal to 1 (≤ 1). If it is, the current subtree is considered balanced, and `isBalanced` is set to true in the `BalanceStatusWithHeight` object. Otherwise, it's unbalanced, and `isBalanced` is set to false.

It calculates the height of the current subtree by finding the maximum height between the left and right subtrees (`Math.max(leftResult.height, rightResult.height)`) and adding 1 to account for the current root node. The result is stored in the height field of the `BalanceStatusWithHeight` object.

- Returning the Balance Status and Height

Finally, the function returns a new `BalanceStatusWithHeight` object that contains the calculated `isBalanced` status and height for the current subtree rooted at t.

Output :

```

-----
# Original tree: Array to Tree #

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

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

```

5. 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.

```

// 5. insertHB() [10 points]
public static MyTree insertHB(int n, MyTree t) {
    if (t.isEmpty()) {
        return new MyTree(n, new MyTree(), new MyTree());
    }
    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;
    }
}

```

This code defines a function to insert a new node with value n into a height-balanced MyTree and ensure the tree remains balanced after the insertion. Here's a breakdown:

- **Public Function: insertHB(int n, MyTree t)**
This function takes an integer n (the value to insert) and a MyTree object t as inputs. It returns a new MyTree object representing the balanced tree after inserting n.
Base Case: Empty Tree

If the tree t is empty (t.isEmpty()), it creates a new MyTree with the value n and empty left and right subtrees. This becomes the new balanced tree after the insertion.

- **Recursive Insertion based on Value Comparison**
If n is less than the value of the root node (t.getValue()), it inserts n into the left subtree. Here's what happens:

A new tree is created with the root's value (`t.getValue()`) as the root, the result of recursively inserting `n` into the left subtree (`insertHB(n, t.getLeft())`) as the left child, and the original right subtree (`t.getRight()`) as the right child.

The resulting tree might become unbalanced due to the insertion on the left. The function calls `rebalanceForLeft` (explained later) to fix the imbalance and return the balanced tree.

If `n` is greater than the value of the root node, it inserts `n` into the right subtree following similar logic:

A new tree is created with the root's value (`t.getValue()`) as the root, the original left subtree (`t.getLeft()`) as the left child, and the result of recursively inserting `n` into the right subtree (`insertHB(n, t.getRight())`) as the right child.

The function calls `rebalanceForRight` (explained later) to address any imbalance caused by the insertion on the right and return the balanced tree.

If `n` is equal to the value of the root node (`t`), it simply returns the original tree `t` as no insertion is needed.

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

```
// 6. rebalanceForLeft() [15 points]
private static MyTree rebalanceForLeft(MyTree t) {
    int balance = getBalance(t);
    if (balance > 1) {
        if (getBalance(t.getLeft()) < 0) {
            // Left-Right Case
            MyTree leftRotated = new MyTree(t.getLeft().getValue(), t.getLeft().getLeft(), rotateLeft(t.getLeft().getRight()));
            return rotateRight(new MyTree(t.getValue(), leftRotated, t.getRight()));
        }
        // Left-Left Case
        return rotateRight(t);
    }
    return t;
}
```

this code defines the `rebalanceForLeft` function, which is responsible for restoring balance in a height-balanced tree after an insertion on the left side.

- Input and Output:

The function takes a `MyTree` object `t` as input, representing the potentially unbalanced tree after a left-side insertion.

It returns a new `MyTree` object representing the balanced tree after rebalancing.

- Balance Calculation:

`getBalance(t)`: It calls a helper function (likely not shown here) named `getBalance` to calculate the balance factor of the current tree `t`. The balance factor indicates the height difference between the left and right subtrees.

- Checking for Imbalance:

balance > 1: The function checks if the balance factor (balance) is greater than 1. This signifies a right-heavy imbalance caused by the insertion on the left.

- Left-Right Case (Double Rotation):

getBalance(t.getLeft()) < 0: If the balance factor of the left child (t.getLeft()) is less than zero, it indicates a further imbalance within the left subtree (left-heavy). This scenario requires a left-right or double rotation.

Creating Left-Rotated Subtree: It constructs a new left subtree (leftRotated) by performing a left rotation on the right child of the left child (t.getLeft().getRight()). This rotation corrects the imbalance within the left subtree.

Performing Right Rotation: Finally, it performs a right rotation on the entire tree with the root t.getValue(), the newly created balanced left subtree (leftRotated), and the original right subtree (t.getRight()). This combined rotation addresses both imbalances (left-heavy within the left subtree and right-heavy overall).

- Left-Left Case (Single Rotation):

else (Left-Left Case): If the balance factor of the left child (t.getLeft()) is greater than or equal to zero (0 or positive), it indicates a left-heavy imbalance within the left subtree itself (left-left case). This can be corrected with a single right rotation.

Right Rotation: The function simply performs a right rotation on the entire tree t. This rotation promotes the left child (which is already heavier) to become the new root and balances the tree.

- Returning the Balanced Tree:

return t: If the balance factor (balance) is not greater than 1 (meaning the tree is already balanced or the imbalance is not severe enough), the function simply returns the original tree t without modification.

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

```

// 7. rebalanceForRight() [15 points]
private static MyTree rebalanceForRight(MyTree t) {
    // Write your codes in here
    int balance = getBalance(t);
    if (balance < -1) {
        if (getBalance(t.getRight()) > 0) {
            // Right-Left Case
            MyTree rightRotated = new MyTree(t.getRight().getValue(), rotateRight(t.getRight().getLeft()), t.getRight().getRight());
            return rotateLeft(new MyTree(t.getValue(), t.getLeft(), rightRotated));
        }
        // Right-Right Case
        return rotateLeft(t);
    }
    return t;
}

```

This code defines the `rebalanceForRight` function, which is responsible for restoring balance in a height-balanced tree after an insertion on the right side. The logic is similar to `rebalanceForLeft` but focuses on the right subtree.

Output for 4,5,6 :

```

Question 5: insertHB()
-----
# 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
-----

```

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.

```
// 8. deleteHB() [10 points]
public static MyTree deleteHB(MyTree t, int x) {
    // Write your codes in here
    if (t.getEmpty()) {
        return t;
    }

    // Step 1: Perform standard BST delete
    if (x < t.getValue()) {
        t = new MyTree(t.getValue(), deleteHB(t.getLeft(), x), t.getRight());
    } else if (x > t.getValue()) {
        t = new MyTree(t.getValue(), t.getLeft(), deleteHB(t.getRight(), x));
    } else {
        // Node with only one child or no child
        if (t.getLeft().getEmpty() || t.getRight().getEmpty()) {
            MyTree temp = t.getLeft().getEmpty() ? t.getRight() : t.getLeft();
            if (temp.getEmpty()) {
                // No child case
                temp = new MyTree();
            }
            return temp;
        } else {
            MyTreeNode temp = minValueNode(t.getRight());
            t = new MyTree(temp.getValue(), t.getLeft(), deleteHB(t.getRight(), temp.getValue()));
        }
    }
    if (t.getEmpty()) {
        return t;
    }
    // Step 2: Rebalance the tree
    return rebalance(t);
}

private static MyTree minValueNode(MyTree t) {
    MyTree current = t;
    while (!current.getLeft().getEmpty()) {
        current = current.getLeft();
    }
    return current;
}

private static MyTree rebalance(MyTree t) {
    int balance = getBalance(t);
    // Left heavy
    if (balance > 1) {
        if (getBalance(t.getLeft()) >= 0) {
            return rotateRight(t);
        } else {
            t = new MyTree(t.getValue(), rotateLeft(t.getLeft()), t.getRight());
            return rotateRight(t);
        }
    }

    // Right heavy
    if (balance < -1) {
        if (getBalance(t.getRight()) <= 0) {
            return rotateLeft(t);
        } else {
            t = new MyTree(t.getValue(), t.getLeft(), rotateRight(t.getRight()));
            return rotateLeft(t);
        }
    }

    return t;
}
```

- Standard BST Deletion:
 - Recursively searches for the node with value x.
 - Handles nodes with one child or no child (replacement not needed).
 - For nodes with two children, finds the in-order successor and replaces the value.
 - Recursively deletes the in-order successor from the right subtree.
- Rebalance (if not empty):
 - Checks balance of the tree after deletion.
 - Performs single or double rotations (depending on imbalance) to restore balance.

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

15 has been deleted

```
5
  |
  - 16
  - 4
```

The tree is Height-Balanced