BuggyPharmacy.java
Code review nao deve demorar mais de 5 minutos.

```
import java.util.*;

// This code creates represents a pharmacy, which has a stock of products that can
be purchased by users.
// The stock class stores the products and their quantities.
// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
// 5. Documentation bug

// The review should take no longer than 5 minutes.
public class BuggyPharmacy {

  // create product class
  public static class Product {
    private String name;
    private double price;
    private String barcode;

    public Product(String name, double price, String barcode) {
      this.name = name;
      this.price = price;
      this.barcode = barcode;
    }

    public String getName() { return name; }

    public double getPrice() { return price; }

    public String getBarcode() { return barcode; }

    @Override
    public String toString() {
      return name + " (Barcode: " + barcode + ", Price: $" + price + ")";
    }
  }

  // create stock class, which has a map of products and their quantities
  public static class Stock {
    private Map<Product, Integer> products;

    public Stock() { this.products = new HashMap<>(); }

    //add a certain quantity of product to stock
    public void addProduct(Product product, int quantity) {
      products.put(product, quantity);
    }

    public void removeProduct(Product product) { products.remove(product); }

    // updates the product quantity on stock
    // Bug #1: Incorrectly updates quantity by ignoring the quantity parameter
    public void updateQuantity(Product product, int quantity) {
      products.put(product, quantity + 1);
```

```
  }

  // purchase products from stock
  // Bug #2: Incorrectly handles purchase, reducing quantity by 1 instead of
  // the quantity bought
  public void purchase(User user, Map<Product, Integer> productsToPurchase) {
    System.out.println(user.getName() +
                       " is purchasing the following products:");
    for (Map.Entry<Product, Integer> productEntry :
         productsToPurchase.entrySet()) {
      int availableQuantity = products.get(productEntry.getKey());
      if (availableQuantity > 0) {
        System.out.println("- " + productEntry.getKey() +
                           " (Quantity: " + productEntry.getValue() + ")");
        updateQuantity(productEntry.getKey(), availableQuantity - 1);
      } else {
        System.out.println("- " + productEntry.getKey() + " (Out of stock)");
      }
    }
  }

  @Override
  public String toString() {
    return "Stock: " + products;
  }
}

// create user class
public static class User {
  private String name;
  private String phone;
  private float fiscalNumber;

  // Bug #3: Incorrectly uses float for fiscal number
  public User(String name, String phone, float fiscalNumber) {
    this.name = name;
    this.phone = phone;
    this.fiscalNumber = fiscalNumber;
  }

  public String getName() { return name; }

  public String getPhone() { return phone; }

  public float getFiscalNumber() { return fiscalNumber; }

  @Override
  public String toString() {
    return "User: " + name + " (Phone: " + phone +
        ", Fiscal Number: " + fiscalNumber + ")";
  }
}

public static void main(String[] args) {
  // Sample products
  Product paracetamol = new Product("Paracetamol", 5.99, "123456");
  Product aspirin = new Product("Aspirin", 3.49, "789012");

  // Sample Stock
```

```
    Stock Stock = new Stock();
    Stock.addProduct(paracetamol, 50);
    Stock.addProduct(aspirin, 30);

    // Sample user
    User customer = new User("John Doe", "+123456789", 39288123);

    Map<Product, Integer> productsToPurchase =
        new HashMap<BuggyPharmacy.Product, Integer>();
    productsToPurchase.put(aspirin, 2);
    productsToPurchase.put(paracetamol, 1);
    Stock.purchase(customer, productsToPurchase);
  }
}
```

BuggyBackpack.java
Review nao deve demorar mais de 10 minutos

```
import java.util.*;
import java.util.stream.Collectors;

// This code creates represents a backpack, which can be packed with items using
different strategies.
// Each item has a type, name and weight in Kg and is unique.
// The backpack class stores the items.
// The backpack can be packed with random items, lightest items first,
// heaviest items first, or with a special rule that packs food items separately.
// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
// 5. Documentation bug

// The review should take no longer than 10 minutes.
public class BuggyBackpack {

  // type of items
  public enum ItemType {
    FOOD, CLOTHING, BOOK, TOY
  }

  // Create item class
  public static class Item {
    private ItemType type;
    private String name;
    private double weight;

    public Item(ItemType type, String name, double weight) {
      this.type = type;
      this.name = name;
      this.weight = weight;
    }
```

```
    public ItemType getType() {
      return type;
    }

    public String getName() {
      return name;
    }

    public double getWeight() {
      return weight;
    }

    @Override
    public String toString() {
      // Bug #1: Displaying item name instead of type
      return name + " (" + name + ", " + weight + " kg)";
    }
  }

  // Packing strategy
  public enum Strategy {
    RANDOM, // Pack random items
    HEAVY_FIRST, // Pack the heaviest items first
    LIGHT_FIRST // Pack the lightest items first
  }

  // Create backpack class
  public static class Backpack {
    private List<Item> items;

    public Backpack() {
      this.items = new ArrayList<>();
    }

    public void addItem(Item item) {
      items.add(item);
    }

    // Sorts items by weight
    public void sortItemsByWeight() {
      items.sort(Comparator.comparingDouble(Item::getWeight));
    }

    // Displays backpack contents
    public void displayContents() {
      System.out.println("Backpack Contents:");
      for (Item item : items) {
        System.out.println("- " + item);
      }
      // Bug #2: Displaying the object address because toString is not implemented
      System.out.println(this);
    }

    // Packs food items separately from non-food items
    // Bug #3: Doesnt check if food items to pack are more than the total items to
    // pack
    public void packItemsWithSpecialRules(List<Item> availableItems, int
totalItemsToPack, int foodItemsToPack) {
      // Simulate a scenario where FOOD items need special handling
```

```
      List<Item> foodItems = availableItems.stream()
          .filter(item -> item.getType() == ItemType.FOOD)
          .collect(Collectors.toList());

      List<Item> nonFoodItems = availableItems.stream()
          .filter(item -> item.getType() != ItemType.FOOD)
          .collect(Collectors.toList());

      // Ensure that FOOD items are packed separately to maintain freshness
      int numFoodItemsToPack = foodItemsToPack;
      int numNonFoodItemsToPack = totalItemsToPack - numFoodItemsToPack;

      packRandomItems(nonFoodItems, numNonFoodItemsToPack);
      packRandomItems(foodItems, numFoodItemsToPack);

      // Display a message indicating the special packing scenario
      System.out.println("Packing items with special rules:");
      System.out.println("Non-FOOD items packed: " + numNonFoodItemsToPack);
      System.out.println("FOOD items packed: " + numFoodItemsToPack);
    }

    // Bug #4: wrong description. It can pack more than one item.
    // packs one item using a strategy
    public void packItems(List<Item> availableItems, Strategy strategy,
        int totalItemsToPack) {
      switch (strategy) {
        case RANDOM:
          packRandomItems(availableItems, totalItemsToPack);
          break;
        case HEAVY_FIRST:
          packHeaviestItemsFirst(availableItems, totalItemsToPack);
          break;
        case LIGHT_FIRST:
          packLightestItemsFirst(availableItems, totalItemsToPack);
          break;
      }
    }

    private void packRandomItems(List<Item> availableItems, int totalItemsToPack) {
      Random random = new Random();
      // Bug #5: not preventing from adding the same item more than once
      for (int i = 0; i < totalItemsToPack; i++) {
        Item randomItem =
availableItems.get(random.nextInt(availableItems.size()));
        addItem(randomItem);
      }
    }

    private void packHeaviestItemsFirst(List<Item> availableItems,
        int totalItemsToPack) {
      // Bug #6: Incorrect sorting logic
      availableItems.sort(Comparator.comparingDouble(Item::getWeight));
      addItemRange(availableItems, totalItemsToPack);
    }

    private void packLightestItemsFirst(List<Item> availableItems,
        int totalItemsToPack) {
      // Bug #7: Incorrect sorting logic
      availableItems.sort(
```

```
          Comparator.comparingDouble(Item::getWeight).reversed());
      addItemRange(availableItems, totalItemsToPack);
    }

    private void addItemRange(List<Item> itemsToAdd, int totalItemsToPack) {
      for (int i = 0; i < itemsToAdd.size(); i++) {
        // Bug #8: Incorrect loop condition. Using size of itemsToAdd
        // when should be using totalItemsToPack (number of items to add defined by
        // caller)
        addItem(itemsToAdd.get(i));
      }
    }
  }

  public static void main(String[] args) {
    Backpack backpack1 = new Backpack();
    Backpack backpack2 = new Backpack();
    Backpack backpack3 = new Backpack();
    Backpack backpack4 = new Backpack();
    List<Item> availableItems = generateAvailableItems();

    // Pack items using different strategies
    backpack1.packItems(availableItems, Strategy.RANDOM, 3);
    backpack2.packItems(availableItems, Strategy.HEAVY_FIRST, 4);
    backpack3.packItems(availableItems, Strategy.LIGHT_FIRST, 4);
    backpack4.packItemsWithSpecialRules(availableItems, 5, 2);

    // Display the contents of the backpack
    backpack1.displayContents();
    backpack2.displayContents();
    backpack3.displayContents();
    backpack4.displayContents();
  }

  private static List<Item> generateAvailableItems() {
    return Arrays.asList(new Item(ItemType.FOOD, "Apple", 0.2),
        new Item(ItemType.FOOD, "Sandwich", 0.5),
        new Item(ItemType.CLOTHING, "T-shirt", 0.3),
        new Item(ItemType.CLOTHING, "Jeans", 0.7),
        new Item(ItemType.BOOK, "Novel", 0.8),
        new Item(ItemType.BOOK, "Notebook", 0.4),
        new Item(ItemType.TOY, "Teddy Bear", 0.6),
        new Item(ItemType.TOY, "Toy Car", 0.3));
  }
}
```

BuggyTree.java
Review nao deve demorar mais de 15 minutos

```
import java.util.Scanner;

// This code creates an AVL Tree and performs operations like insertion,
// deletion, searching, etc. The AVL tree is a self-balancing binary search
// tree. In an AVL tree, the heights of the two child subtrees of any node
// differ by at most one.
```

```
// If the tree only has one node, the height of the tree is 0.
// Lack of handling of user input errors doesnt count as a bug.

// Bugs can be of the following types:
// 1. Logical bug
// 2. Performance bug
// 3. Variable type bug
// 4. Memory managment bug
// 5. Documentation bug

// The review should take no longer than 15 minutes.

//  create Node class to design the structure of the AVL Tree Node
class Node {
    int element;
    int h; // for height
    Node leftChild;
    Node rightChild;

    // default constructor to create null node
    public Node() {
        leftChild = null;
        rightChild = null;
        element = 0;
        h = 0;
    }

    // parameterized constructor
    public Node(int element) {
        leftChild = null;
        rightChild = null;
        this.element = element;
        h = 0;
    }
}

// create class ConstructAVLTree for constructing AVL Tree
class ConstructAVLTree {
    private Node rootNode;

    // Constructor to set null value to the rootNode
    public ConstructAVLTree() {
        rootNode = null;
    }

    // create removeAll() method to make AVL Tree empty
    public void removeAll() {
        rootNode = null;
    }

    // create checkEmpty() method to check whether the AVL Tree is empty or not
    public boolean checkEmpty() {
        if (rootNode == null)
            return true;
        else
            return false;
    }

    // create insertElement() to insert element to to the AVL Tree
```

```
    public void insertElement(int element) {
        rootNode = insertElement(element, rootNode);
    }

    // create getHeight() method to get the height of the AVL Tree
    // Bug #1: Incorrectly calculates height of tree. Should be -1 when tree is
    // empty
    private int getHeight(Node node) {
        return node == null ? 0 : node.h;
    }

    // create maxNode() method to get the maximum height from left and right node
    // Bug #2: Incorrectly calculates max height. Should be > comparison instead
    // of >=
    private int getMaxHeight(int leftNodeHeight, int rightNodeHeight) {
        return leftNodeHeight >= rightNodeHeight ? leftNodeHeight :
rightNodeHeight;
    }

    // create insertElement() method to insert data in the AVL Tree recursively
    private Node insertElement(int element, Node node) {
        // check whether the node is null or not
        if (node == null)
            node = new Node(element);
        // insert a node in case when the given element is lesser than the element
        // of the root node
        else if (element < node.element) {
            node.leftChild = insertElement(element, node.leftChild);
            if (getHeight(node.leftChild) - getHeight(node.rightChild) == 2)
                if (element < node.leftChild.element)
                    node = rotateWithLeftChild(node);
                else
                    node = doubleWithLeftChild(node);
        }
        // Bug #3: Incorrectly documentation. Should be greater than instead of
lesser
        // than.
        // insert a node in case when the given element is lesser than the element
        // of the root node
        else if (element > node.element) {
            node.rightChild = insertElement(element, node.rightChild);
            if (getHeight(node.rightChild) - getHeight(node.leftChild) == 2)
                if (element > node.rightChild.element)
                    node = rotateWithRightChild(node);
                else
                    node = doubleWithRightChild(node);
        } else
            ; // if the element is already present in the tree, we will do nothing
        node.h = getMaxHeight(getHeight(node.leftChild),
getHeight(node.rightChild)) + 1;

        return node;
    }

    // creating rotateWithLeftChild() method to perform rotation of binary tree
    // node with left child
    private Node rotateWithLeftChild(Node node2) {
        Node node1 = node2.leftChild;
        node2.leftChild = node1.rightChild;
```

```
        node1.rightChild = node2;
        node2.h = getMaxHeight(getHeight(node2.leftChild),
getHeight(node2.rightChild)) +
                1;
        node1.h = getMaxHeight(getHeight(node1.leftChild), node2.h) + 1;
        return node1;
    }

    // creating rotateWithRightChild() method to perform rotation of binary tree
    // node with right child
    private Node rotateWithRightChild(Node node1) {
        Node node2 = node1.rightChild;
        node1.rightChild = node2.leftChild;
        node2.leftChild = node1;
        node1.h = getMaxHeight(getHeight(node1.leftChild),
getHeight(node1.rightChild)) +
                1;
        node2.h = getMaxHeight(getHeight(node2.rightChild), node1.h) + 1;
        return node2;
    }

    // create doubleWithLeftChild() method to perform double rotation of binary
    // tree node. This method first rotate the left child with its right child,
    // and after that, node3 with the new left child
    private Node doubleWithLeftChild(Node node3) {
        node3.leftChild = rotateWithRightChild(node3.leftChild);
        return rotateWithLeftChild(node3);
    }

    // create doubleWithRightChild() method to perform double rotation of binary
    // tree node. This method first rotate the right child with its left child and
    // after that node1 with the new right child
    private Node doubleWithRightChild(Node node1) {
        node1.rightChild = rotateWithLeftChild(node1.rightChild);
        return rotateWithRightChild(node1);
    }

    // create getTotalNumberOfNodes() method to get total number of nodes in the
    // AVL Tree
    public int getTotalNumberOfNodes() {
        return getTotalNumberOfNodes(rootNode);
    }

    private int getTotalNumberOfNodes(Node head) {
        if (head == null)
            return 0;
        else {
            int length = 1;
            length = length + getTotalNumberOfNodes(head.leftChild);
            length = length + getTotalNumberOfNodes(head.rightChild);

            // #Bug 4: Unnecessary work - Recalculating the height for the entire
subtree.
            int leftHeight = getHeight(head.leftChild);
            int rightHeight = getHeight(head.rightChild);
            getMaxHeight(leftHeight, rightHeight);
            return length;
        }
    }
```

```
// create searchElement() method to find an element in the AVL Tree
public boolean searchElement(int element) {
    return searchElement(rootNode, element);
}

// Bug #5: Logic bug. Recursive call not necessary here, since the loop
// already traverses the tree
private boolean searchElement(Node head, int element) {
    boolean check = false;
    while ((head != null) && !check) {
        int headElement = head.element;
        if (element < headElement)
            head = head.leftChild;
        else if (element > headElement)
            head = head.rightChild;
        else {
            check = true;
            break;
        }
        check = searchElement(head, element);
    }
    return check;
}

// create inorderTraversal() method for traversing AVL Tree in in-order form
public void inorderTraversal() {
    inorderTraversal(rootNode);
}

private void inorderTraversal(Node head) {
    if (head != null) {
        inorderTraversal(head.leftChild);
        System.out.print(head.element + " ");
        inorderTraversal(head.rightChild);
    }
}

// create preorderTraversal() method for traversing AVL Tree in pre-order form
public void preorderTraversal() {
    preorderTraversal(rootNode);
}

private void preorderTraversal(Node head) {
    if (head != null) {
        System.out.print(head.element + " ");
        preorderTraversal(head.leftChild);
        preorderTraversal(head.rightChild);
    }
}

// create postorderTraversal() method for traversing AVL Tree in post-order
// form
public void postorderTraversal() {
    postorderTraversal(rootNode);
}

private void postorderTraversal(Node head) {
    if (head != null) {
```

```
            postorderTraversal(head.leftChild);
            postorderTraversal(head.rightChild);
            System.out.print(head.element + " ");
        }
    }
}

// create AVLTree class to construct AVL Tree
public class BuggyTree {
    // main() method starts
    public static void main(String[] args) {
        // creating Scanner class object to get input from user
        Scanner sc = new Scanner(System.in);

        // create object of ConstructAVLTree class object for costructing AVL Tree
        ConstructAVLTree obj = new ConstructAVLTree();

        char choice; // initialize a character type variable to choice

        // perform operation of AVL Tree using switch
        do {
            System.out.println("\nSelect an operation:\n");
            System.out.println("1. Insert a node");
            System.out.println("2. Search a node");
            System.out.println("3. Get total number of nodes in AVL Tree");
            System.out.println("4. Is AVL Tree empty?");
            System.out.println("5. Remove all nodes from AVL Tree");
            System.out.println("6. Display AVL Tree in Post order");
            System.out.println("7. Display AVL Tree in Pre order");
            System.out.println("8. Display AVL Tree in In order");

            // get choice from user
            int ch = sc.nextInt();
            switch (ch) {
                case 1:
                    System.out.println("Please enter an element to insert in AVL
Tree");
                    obj.insertElement(sc.nextInt());
                    break;
                case 2:
                    System.out.println("Enter integer element to search");
                    System.out.println(obj.searchElement(sc.nextInt()));
                    break;
                case 3:
                    System.out.println(obj.getTotalNumberOfNodes());
                    break;
                case 4:
                    System.out.println(obj.checkEmpty());
                    break;
                case 5:
                    obj.removeAll();
                    System.out.println("\nTree Cleared successfully");
                    break;
                case 6:
                    System.out.println("\nDisplay AVL Tree in Post order");
                    obj.postorderTraversal();
                    break;
                case 7:
                    System.out.println("\nDisplay AVL Tree in Pre order");
```

```
                    obj.preorderTraversal();
                    break;
                case 8:
                    System.out.println("\nDisplay AVL Tree in In order");
                    obj.inorderTraversal();
                    break;
                default:
                    System.out.println("\n ");
                    break;
            }
            System.out.println("\nPress 'y' or 'Y' to continue \n");
            choice = sc.next().charAt(0);
        } while (choice == 'Y' || choice == 'y');

        sc.close();
    }
}
```