COMPLEXITY

US08

```
/**
 * Builds the production tree with the specified main objective.
 * @param mainObjectiveID the ID of the main objective
 */
public TreeNode<String> buildProductionTree(
        String mainObjectiveID
) {
    BOORepository booRepository =
Instances.getInstance().getBOORepository();
    ItemsRepository itemsRepository =
Instances.getInstance().getItemsRepository();
    OperationsMapRepository operationsMapRepository =
Instances.getInstance().getOperationsMapRepository();
    List<String[]> booData = booRepository.getBOORepository();
    Map<String, String> itemNames = itemsRepository.getItemsRepository();
    Map<String, String> operationDescriptions =
operationsMapRepository.getOperationsMapRepository();

    // Identifies the operation associated with mainObjectiveID in BOO
    String initialOperationID = null;
    String quantityOperation = null;

    for (String[] entry : booData) {
        if (entry.length >= 2 && entry[1].equals(mainObjectiveID)) {
            initialOperationID = entry[0];
            quantityOperation = entry[2];
            break;
        }
    }

    if (initialOperationID == null) {
        System.out.println("Main objective not found in the Bill of
Operations.");
        return null; // Returns null if the main objective is not found
    }

    // Get the description of the initial operation
    String mainOperationDescription =
operationDescriptions.getOrDefault(initialOperationID, "Unknown
Operation");

    // Create the root node as the main operation
    root = new TreeNode<>(mainOperationDescription + " (Quantity: " +
quantityOperation + ")", NodeType.OPERATION);

    // Build the production tree recursively
    buildSubTree(initialOperationID, root, booData, itemNames,
operationDescriptions);

    return root; // Returns the complete tree
}
```

Time Complexity buildProductionTree(): O(n * m)

- The outer loop over booData has a complexity of **O(n),** where n is the number of entries.
- The recursive tree construction processes each node and its children, leading to **O(m)** operations per node, where m is the maximum number of child nodes.
- Combining both, the overall complexity is **O (n * m).**

```
/**
 * Builds the subtree of the production tree recursively.
 * @param currentOperationID the ID of the current operation
 * @param parent the parent node of the current operation
 * @param booData the data from the Bill of Operations
 * @param itemNames the map of item IDs to item names
 * @param operationDescriptions the map of operation IDs to operation
descriptions
 */
private void buildSubTree(
        String currentOperationID,
        TreeNode<String> parent,
        List<String[]> booData,
        Map<String, String> itemNames,
        Map<String, String> operationDescriptions
) {
    for (String[] booEntry : booData) {
        if (booEntry.length >= 2 && booEntry[0].equals(currentOperationID))
{
            String productID = booEntry[1];
            String productQuantity = booEntry[2];

            String productName = itemNames.getOrDefault(productID, "Unknown
Product");
            TreeNode<String> productNode = new TreeNode<>(productName + "
(Quantity: " + productQuantity + ")", NodeType.MATERIAL);
            productNode.setOperationParent(parent); // Define the parent
            parent.addChild(productNode);

            nodesMap.put(productID, productNode);

            // Add sub-operations
            int numberOperations = countOperations(booEntry);
            int k = 4;
            while (k < 4 + 2 * numberOperations) {
                String subOperationId = booEntry[k];
                String subOperationQuantity = booEntry[k + 1];
                k += 2;

                String subOperationDescription =
operationDescriptions.getOrDefault(subOperationId, "Unknown Operation");
                TreeNode<String> subOperationNode = new
TreeNode<>(subOperationDescription + " (Quantity: " + subOperationQuantity
+ ")");
                subOperationNode.setType(NodeType.OPERATION);
                subOperationNode.setOperationParent(parent); // Define the
parent
```

```
                productNode.addChild(subOperationNode);

                int depth = calculateDepth(subOperationNode);
                int priority = getPriorityForDepth(depth);
                qualityCheckQueue.insert(priority,
subOperationDescription);

                nodesMap.put(subOperationId, subOperationNode);

                buildSubTree(subOperationId, subOperationNode, booData,
itemNames, operationDescriptions);
            }

            // Add Materials
            int materialsStartIndex = findMaterialsStartIndex(booEntry);
            int numberMaterials = countMaterials(booEntry);
            for (int j = materialsStartIndex; j < materialsStartIndex + 2 *
numberMaterials; j += 2) {
                String materialId = booEntry[j];
                String quantity = booEntry[j + 1];
                String materialName = itemNames.getOrDefault(materialId,
"Unknown Material");

                TreeNode<String> materialNode = new TreeNode<>(materialName
+ " (Quantity: " + quantity + ")");
                materialNode.setType(NodeType.MATERIAL);
                materialNode.setOperationParent(parent); // Define the
parent
                productNode.addChild(materialNode);

                nodesMap.put(materialId, materialNode);

                buildSubTree(materialId, materialNode, booData, itemNames,
operationDescriptions);
            }
        }
    }
}
```

Time Complexity buildSubTree: O (n * m)

- The method iterates over the booData list, with complexity **O(n),** where n is the number of entries.
- For each entry, it processes numberOperations sub-operations and numberMaterials materials.
    - Handling sub-operations and materials involves loops, with a total complexity of **O(m)** per entry, where m is the maximum number of sub-operations and materials combined for a single node.
- Recursive calls occur for every operation and material, but each node is visited exactly once.
- Overall, the time complexity is **O (n * m).**

```java
/**
 * Finds the indices of the parentheses in the input array.
 * @param inputArray the array to search for parentheses
 * @param startIndex the index to start searching from in the array
 * @return an array with the indices of the opening and closing parentheses
 */
private int[] findParenthesesIndices(String[] inputArray, int startIndex) {
    int openParenIndex = -1;
    int closeParenIndex = -1;

    // Find the index of '('
    for (int i = startIndex; i < inputArray.length; i++) {
        if (inputArray[i].contains("(")) {
            openParenIndex = i;
            break;
        }
    }

    // Find the index of ')'
    for (int i = openParenIndex + 1; i < inputArray.length; i++) {
        if (inputArray[i].contains(")")) {
            closeParenIndex = i;
            break;
        }
    }

    return new int[]{openParenIndex, closeParenIndex};
}
```

Time Complexity findParenthesesIndices: O(n)

- The method performs two independent loops over the inputArray:
    - The first loop searches for the opening parenthesis '(', starting from startIndex.
    - The second loop starts from the position after the opening parenthesis and searches for the closing parenthesis ')'.
- Each loop can iterate through the entire remaining portion of the array in the worst case, making the overall complexity **O(n),** where n is the length of inputArray.

```java
/**
 * Counts the number of elements between parentheses in the input array.
 * @param inputArray the array to count the elements from between
parentheses
 * @param startIndex the index to start counting from in the array
 * @return the number of elements between parentheses
 */
private int countElementsBetweenParentheses(String[] inputArray, int
startIndex) {
    int[] indices = findParenthesesIndices(inputArray, startIndex);
    int openParenIndex = indices[0];
    int closeParenIndex = indices[1];
```

```
    if (openParenIndex == -1 || closeParenIndex == -1) {
        return 0; // Brackets not found
    }

    // Count the elements between the brackets
    int count = 0;
    for (int i = openParenIndex + 1; i < closeParenIndex; i++) {
        String element = inputArray[i].trim();
        if (!element.isEmpty()) {
            count++;
        }
    }

    // Divide by 2 (each element consists of ID and quantity)
    return count / 2;
}
```

Time Complexity countElementsBetweenParentheses: O(n)

- The method calls findParenthesesIndices, which has a complexity of **O(n)** as it scans the array twice to find the indices of '(' and ')'.
- After determining the indices, it iterates over the elements between them. In the worst case, this second loop can iterate over the entire array, but since it processes only a subset of the array, its complexity is bounded by O(n).
- Combining these operations, the overall time complexity remains **O(n)**.

```
/**
 * Finds the index of the start of the materials section in the input
array.
 * @param inputArray the array to search for the materials section
 * @return the index of the start of the materials section
 */
private int findStartIndexAfterSecondParentheses(String[] inputArray) {
    int[] firstPair = findParenthesesIndices(inputArray, 0);
    int[] secondPair = findParenthesesIndices(inputArray, firstPair[1] +
1);

    return (secondPair[0] != -1 && secondPair[0] + 1 < inputArray.length)
            ? secondPair[0] + 1
            : -1;
}
```

Time Complexity findStartIndexAfterSecondParentheses: O(n)

- **First Call to findParenthesesIndices:** This scans the array starting from index 0 to find the first pair of parentheses. This operation has a complexity of **O(n)**.
- **Second Call to findParenthesesIndices:** This scans the remaining part of the array (starting after the first closing parenthesis). In the worst case, it may iterate through the rest of the array, making it also **O(n)**.

- Since the two calls to findParenthesesIndices operate sequentially and not nested, the total complexity is **O(n)**.

```
/**
 * Counts the number of operations in the input array.
 * @param inputArray the array to count the operations from
 * @return the number of operations in the input array
 */
private int countOperations(String[] inputArray) {
    return countElementsBetweenParentheses(inputArray, 0);
}
```

Time Complexity countOperations: O(n)

- The method calls countElementsBetweenParentheses, which itself involves scanning the array to locate the parentheses and count elements between them. This has a complexity of **O(n)**.
- Since countOperations only calls this method and performs no additional operations, its overall complexity is **O(n)**.

```
/**
 * Counts the number of materials in the input array.
 * @param inputArray the array to count the materials from
 * @return the number of materials in the input array
 */
private int countMaterials(String[] inputArray) {
    int[] firstPair = findParenthesesIndices(inputArray, 0);
    return countElementsBetweenParentheses(inputArray, firstPair[1] + 1);
}
```

Time Complexity countMaterials: O(n)

- The method first calls findParenthesesIndices, which searches for parentheses and has a time complexity of **O(n)**.
- After finding the indices, it calls countElementsBetweenParentheses, which also scans the array for elements between the parentheses, contributing an additional **O(n)**.
- Since both operations are sequential, the overall complexity remains **O(n)**.

```
/**
 * Finds the index of the start of the materials section in the input
array.
 * @param inputArray the array to search for the materials section
 * @return the index of the start of the materials section
 */
private int findMaterialsStartIndex(String[] inputArray) {
```

```
    return findStartIndexAfterSecondParentheses(inputArray);
}
```

Time Complexity findMaterialsStartIndex: O(n)

- The method calls findStartIndexAfterSecondParentheses, which in turn calls findParenthesesIndices and processes the array twice. Each of these operations takes **O(n)** time, where n is the size of the input array.
- Therefore, the overall time complexity is **O(n)**.

US09

```
/**
 * Searches for a node in the production tree by its ID or name.
 * @param idOrName the ID or name of the operation or material to search
for
 * @return a map with details such as type, quantity (for materials), and
parent operation if applicable
 */
public Map<String, String> searchNode(String idOrName) {
    Map<String, String> result = new HashMap<>();

    // Find the node with the specified ID or name
    TreeNode<String> node = nodesMap.get(idOrName);
    if (node == null) {
        result.put("Error", "Leaf not found on Production Tree.");
        return result;
    }

    String value = node.getValue();
    NodeType type = node.getType();

    // Determines the type based on NodeType
    if (type == NodeType.OPERATION) {
        result.put("Type", "Operation");
        result.put("Description", value);

        // Find the parent operation directly
        TreeNode<String> parentOperation = node.getOperationParent();
        if (parentOperation != null) {
            result.put("Parent Operation", parentOperation.getValue());
        }else {
            result.put("Parent Operation", "None");
        }

    } else if (type == NodeType.MATERIAL) {
        result.put("Type", "Material");
        result.put("Description", value);

        // Extract the quantity from the material
        String quantity = extractQuantityFromMaterial(value);
        result.put("Quantity", quantity);

        // Find the material's parent operation
        TreeNode<String> parentOperation = node.getOperationParent();
```

```
        if (parentOperation != null) {
            result.put("Parent Operation", parentOperation.getValue());
        }
    }

    return result;
}
```

Time Complexity searchNode: O(1)

- The method performs a direct lookup in the nodesMap using the provided idOrName. This operation takes **O(1)** time due to the hash map's constant time complexity for lookups.
- Once the node is found, the method performs basic string manipulations (like put and get operations) and checks for the parent node, which are all **O(1)** operations.
- Therefore, the overall time complexity is **O(1)**.

```
/**
 * Extracts the quantity from a material string.
 * @param material the material string to extract the quantity from
 * @return the quantity of the material
 */
private String extractQuantityFromMaterial(String material) {
    String[] parts = material.split(" ");
    if (parts.length > 0) {
        return parts[parts.length - 1];
    }
    return "Unknown quantity";
}
```

Time Complexity extractQuantityFromMaterial: O(m)

- The method splits the input string material using the split(" ") function, which processes the entire string and creates an array of parts based on spaces.
- The time complexity of split(" ") is **O(m)**, where m is the length of the input string material.
- After splitting, the method accesses the last element of the resulting array, which is an **O(1)** operation.
- Therefore, the overall time complexity is **O(m)**, where m is the length of the material string.

US10

```java
public List<Map.Entry<Material, Double>> getMaterialQuantityPairs() {
    List<Map.Entry<Material, Double>> materialQuantityPairs = new ArrayList<>();
    for (Map.Entry<String, TreeNode<String>> entry : nodesMap.entrySet()) {
        TreeNode<String> node = entry.getValue();
        if (node.getType() == NodeType.MATERIAL) {
            String value = node.getValue();
            int startIndex = value.indexOf("(Quantity: ");
            int endIndex = value.indexOf(')', startIndex);
            if (startIndex != -1 && endIndex != -1 && startIndex < endIndex) {
                String materialID = entry.getKey();
                String materialName = value.substring(0, startIndex).trim();
                String quantityStr = value.substring(startIndex + 11, endIndex).trim().replace(',', '.');
                double quantity = Double.parseDouble(quantityStr);
                Material material = new Material(materialID, materialName, quantityStr);
                if (materialQuantityPairs.contains(material)) {
                    for (Map.Entry<Material, Double> pair : materialQuantityPairs) {
                        if (pair.getKey().equals(material)) {
                            pair.setValue(pair.getValue() + quantity);
                        }
                    }
                } else {
                    materialQuantityPairs.add(new AbstractMap.SimpleEntry<>(material, quantity));
                }
            }
        }
    }
    return materialQuantityPairs;
}

public void printMaterialQuantitiesInAscendingOrder() {
    MaterialsBST materialQuantityBST = new MaterialsBST();
    List<Map.Entry<Material, Double>> materialQuantityPairs = getMaterialQuantityPairs();
    for (Map.Entry<Material, Double> pair : materialQuantityPairs) {
        List<String> materialNames = new ArrayList<>();
        materialNames.add(pair.getKey().getName());
        MaterialsBST.insert(materialNames, pair.getValue());
    }
    materialQuantityBST.inorder();
}
public void printMaterialQuantitiesInDescendingOrder() {
    MaterialsBST materialQuantityBST = new MaterialsBST();
    List<Map.Entry<Material, Double>> materialQuantityPairs = getMaterialQuantityPairs();
    for (Map.Entry<Material, Double> pair : materialQuantityPairs) {
        List<String> materialNames = new ArrayList<>();
        materialNames.add(pair.getKey().getName());
        MaterialsBST.insert(materialNames, pair.getValue());
    }
    materialQuantityBST.reverseInorder();
}
```

Time complexity getMaterialsQuantitiesPair(): **O(n^2)**

- Loop over nodesMap is O(n) because it as n entries and the complexity inside the loop is O(1).
- The list is searched to check if the material already exists, in the worst case the search is O(m) with m being the number of elements in the list. This would happen for each element, resulting in an overall complexity of O(n m). Since the number of materials can be n, the worst case time complexity is $O(n^2)$.

Time complexity of printMaterialQuantitiesInAscendingOrder(): **$O(n^2)$**

- The function getMaterialsQuantitiesPair() is called so the complexity after the call is $O(n^2)$.
- Inserting in the BST is O(log k), where k is the current number of nodes in the tree. Since its inserted n elements then the time complexity is O(n log n).
- The InOrder takes O(n) time since every node is visited once.
- Since $O(n^2)$ is bigger than O(n log n) the time complexity is $O(n^2)$.

Time complexity of printMaterialQuantitiesInDescendingOrder(): **$O(n^2)$**

- Equal to printMaterialQuantitiesInDescendingOrder(), with the small change that instead of InOrder it is reverseInOrder but both have the same time complexity, O(n).

US11

```
public void viewQualityChecksInOrder() {
    HeapPriorityQueue<Integer, String> tempQueue = qualityCheckQueue.clone();
    System.out.println("Quality Checks in Order of Priority:");
    while (!tempQueue.isEmpty()) {
        var check = tempQueue.removeMin();
        System.out.println("Quality Check: " + check.getValue() + " [Priority: " +
check.getKey() + "]");
    }
}
```

Time Complexity: O(n log n)

- Cloning the priority queue takes O(n)
- Each removeMin operation takes O(log n)
- We perform removeMin n times
- Total: O(n) + O(n log n) = O(n log n)

```
public void performQualityChecksInteractively() {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Starting Interactive Quality Checks:");
    while (!qualityCheckQueue.isEmpty()) {
        var nextCheck = qualityCheckQueue.removeMin();
        System.out.println("Next Quality Check: " + nextCheck.getValue() + "
[Priority: " + nextCheck.getKey() + "]");
        System.out.print("Perform this quality check? (yes/no): ");
        String input = scanner.nextLine().trim().toLowerCase();
        if (input.equals("yes")) {
            System.out.println("Performing Quality Check: " +
nextCheck.getValue());
        } else if (input.equals("no")) {
            System.out.println("Skipping Quality Check: " + nextCheck.getValue());
        } else {
            System.out.println("Invalid input. Skipping Quality Check.");
        }
        System.out.print("Do you want to continue with the next quality check?
(yes/no): ");
         input = scanner.nextLine().trim().toLowerCase();
        if (input.equals("no")) {
            System.out.println("Stopping Quality Checks.");
            break;
        }
    }
    if (qualityCheckQueue.isEmpty()) {
        System.out.println("All Quality Checks have been completed.");
    }
}
```

Time Complexity: O(n log n)

- Each iteration removes one element from the priority queue using removeMin()
- Total complexity is O(n log n) where n is the number of quality checks

US12

```
public void updateQuantities(String materialID, double newQuantity) {
    TreeNode<String> node = nodesMap.get(materialID);
    if (node == null) {
        System.out.println("Material not found in the production tree.");
        return;
    }

    String value = node.getValue();
    int startIndex = value.indexOf("(Quantity: ");
    int endIndex = value.indexOf(')', startIndex);
    if (startIndex != -1 && endIndex != -1 && startIndex < endIndex) {
        String materialName = value.substring(0, startIndex).trim();
        String quantityStr = value.substring(startIndex + 11, endIndex).trim().replace(',', '.');
        double oldQuantity = Double.parseDouble(quantityStr);
        double difference = newQuantity - oldQuantity;
        value = materialName + " (Quantity: " + newQuantity + ")";
        node.setValue(value);
        System.out.println("Updated quantity for " + materialName + " from " + oldQuantity + " to " +
newQuantity);
    }
}
```

Time complexity: O(n) where n is the length of the material description string.

- Fetching the node in nodesMap.get(materialID) is O(1)
- value.indexOf("(Quantity: ") and value.indexOf(')', startIndex) are searching for specific characters in a string. In the worst case, finding the index involves scanning the entire string, which takes O(n), where n is the length of the string.
- Value.substring(...) is also O(n) because it depends on the string they are working with.

US13

```
public Map<String, Object>
calculateTotalMaterialsAndOperations(TreeNode<String> root) {
    Map<String, Double> materialQuantities = new HashMap<>();
    Map<String, Double> operationTimes = new HashMap<>();
    calculateTotals(materialQuantities, operationTimes, root);

    Map<String, Object> result = new HashMap<>();
```

```
    result.put("materialQuantities", materialQuantities);
    result.put("operationTimes", operationTimes);
    return result;
}
```

- Time: O(1) - just creates maps and delegates work
- Space: O(M + O) - creates result map with material and operation maps inside where M = unique materials, O = unique operations

```
public void calculateTotals(Map<String, Double> materialQuantities,
Map<String, Double> operationQuantities, TreeNode<String> root) {
    traverseTree(root, materialQuantities, operationQuantities);
}
```

- Time: O(1) - simply calls traverseTree
- Space: O(1) - no additional space used, just passes references

```
public void traverseTree(TreeNode<String> node, Map<String, Double>
materialQuantities, Map<String, Double> operationQuantities) {
    if (node == null) {
        return;
    }

    String value = node.getValue();
    if (node.getType().equals(NodeType.MATERIAL)) {
        int startIndex = value.indexOf("(Quantity: ");
        int endIndex = value.indexOf(')', startIndex);
        if (startIndex != -1 && endIndex != -1 && startIndex < endIndex) {
            String materialName = value.substring(0, startIndex).trim();
            String quantityStr = value.substring(startIndex + 11,
endIndex).trim().replace(',', '.');
            double quantity = Double.parseDouble(quantityStr);
            materialQuantities.put(materialName,
materialQuantities.getOrDefault(materialName, 0.0) + quantity);
        }
    } else if (node.getType().equals(NodeType.OPERATION)) {
        int startIndex = value.indexOf("(Quantity: ");
        int endIndex = value.indexOf(')', startIndex);
        if (startIndex != -1 && endIndex != -1 && startIndex < endIndex) {
            String operationName = value.substring(0, startIndex).trim();
            String quantityStr = value.substring(startIndex + 11,
endIndex).trim().replace(',', '.');
            double quantity = Double.parseDouble(quantityStr);
            operationQuantities.put(operationName,
operationQuantities.getOrDefault(operationName, 0.0) + quantity);
        }
    }

    for (TreeNode<String> child : node.getChildren()) {
        traverseTree(child, materialQuantities, operationQuantities);
    }
}
```

- Time: O(N * S) - visits all N nodes, performs string operations of length S at each

- Space: O(H) - recursion stack space where H is tree height
- At each node:
    - String parsing operations like indexOf/substring are O(S)
    - HashMap operations getOrDefault/put are O(1)
    - Recursion for children processes each node exactly once

Overall complexity: O(N * S)

- Where N is the total number of nodes in the tree
- S is the average length of string value in each node
- This comes from traverseTree() being the dominant function since it does all the actual work
- The other methods just delegate to it

US14

```
public void prioritizeCriticalPath(TreeNode<String> root) {
    if (root == null) {
        System.out.println("Production tree is empty.");
        return;
    }

    // Priority Queue to store operations by depth
    HeapPriorityQueue<Integer, TreeNode<String>> criticalPathQueue = new
HeapPriorityQueue<>();

    // Recursive function to traverse and calculate depth
    traverseAndAddToHeap(root, criticalPathQueue);

    // Display the critical path in order
    System.out.println("Critical Path (in order of the most important to the least
important):");
    while (!criticalPathQueue.isEmpty()) {
        Entry<Integer, TreeNode<String>> entry = criticalPathQueue.removeMin();
        TreeNode<String> node = entry.getValue();
        System.out.println("Operation: " + node.getValue() + " (Depth: " + -
entry.getKey() + ")");
    }
}
```

Time Complexity: O(n log n)

- traverseAndAddToHeap visits each node once: O(n)
- For each node, we might perform an insert operation on the heap: O(log n)

- The while loop performs n removeMin operations: O(n log n)
- Total: O(n log n)

```
private void traverseAndAddToHeap(TreeNode<String> node,
HeapPriorityQueue<Integer, TreeNode<String>> queue) {
    if (node.getType() == NodeType.OPERATION) {
        int depth = calculateDepth(node);
        // Use negative depth to simulate max-heap behavior
        queue.insert(-depth, node);
    }
    for (TreeNode<String> child : node.getChildren()) {
        traverseAndAddToHeap(child, queue);
    }
}
```

Time Complexity: O(n log n)

- Visits each node once: O(n)
- For each operation node:
    - calculateDepth(): O(h) where h is the height of the tree
    - queue.insert(): O(log n)
- Total: O(n * (h + log n))
- In worst case where tree is unbalanced, h = n, making it $O(n^2)$
- In balanced tree case, h = log n, making it O(n log n)

```
public void traverseCriticalPath(TreeNode<String> node) {
    if (node == null) return;

    // Perform a reverse traversal of the children first
    for (TreeNode<String> child : node.getChildren()) {
        traverseCriticalPath(child);
    }

    // Visit the current node
    if (node.getType() == NodeType.OPERATION) {
        System.out.println(node.getValue());
    }
}
```

Time Complexity: O(n)

- Simple post-order traversal
- Visits each node exactly once
- No additional data structures or complex operations

US15

```
public LinkedHashMap<String, Double> simulateBOMBOO() {
    ProductionTree productionTree =
Instances.getInstance().getProductionTree();
    TreeNode<String> root = productionTree.getRoot();
    bombooTree = new AVL<>();
    LinkedHashMap<Integer, BOO> materials = new LinkedHashMap<>();
    List<BOO> postOrderElements = createBOMBOOTree(root, materials);
    timeOperations = new LinkedHashMap<>();
    WorkstationRepository workstationRepository =
Instances.getInstance().getWorkstationRepository();
    HashMap<Integer, Workstation> workstationsMap = (HashMap<Integer,
Workstation>) workstationRepository.getWorkstations();
    LinkedHashMap<String, LinkedList<Material>> operationsQueue = new
LinkedHashMap<>();
    ArrayList<Workstation> workstations = new
ArrayList<>(workstationsMap.values());
    removeNullMachines(workstations);
    ArrayList<Material> filteredMaterials = new ArrayList<>();
    fillItemsWithMaterials(materials, filteredMaterials);
    fillOperationsQueue(filteredMaterials, operationsQueue,
postOrderElements);
    fillUpMachinesUS16(operationsQueue, workstations, filteredMaterials);
    return SimulatorResetUS16(workstations, filteredMaterials);
}
```

- Time Complexity: O(n * m * k)
  - n = number of materials
  - m = number of operations
  - k = number of workstations
- The complexity is dominated by the nested operations in fillUpMachinesUS16()

```
private List<BOO> createBOMBOOTree(TreeNode<String>
node,LinkedHashMap<Integer, BOO> materials) {
    BOO root = new BOO();
    createBOMBOOTree(node, root);
    List<BOO> postOrder = bombooTree.getAllNodes();
    Iterable<BOO> postOrderElements = changeOrder(postOrder);
    fillMaterials(materials, postOrderElements);
    List<BOO> postOrderList = new ArrayList<>();
    postOrderElements.forEach(postOrderList::add);
    return postOrderList;
}
```

- Time Complexity: O(n)
  - n = number of nodes in the tree
- Consists of:
  - Tree traversal and creation: O(n)
  - Getting all nodes: O(n)
  - Changing order: O(n)

    - Filling materials: O(n)
- While there are multiple O(n) operations, they are sequential, not nested

```
private void createBOMBOOTree(TreeNode<String> node, BOO firstElement) {
    if (node == null) {
        return;
    }
    // Process the current node
    String value = node.getValue();
    int startIndex = value.indexOf("(Quantity: ");
    int endIndex = value.indexOf(')', startIndex);

    if (startIndex != -1 && endIndex != -1) {
        String name = value.substring(0, startIndex).trim();
        String quantityStr = value.substring(startIndex + 11,
endIndex).trim().replace(',', '.');
        double quantity = Double.parseDouble(quantityStr);
        if (node.getType() == NodeType.OPERATION) {
            if (bombooTree.getLatestInsertedNode() != null) {
                BOO operation = new BOO(quantity, name);
                bombooTree.insert(operation);
                BOO insertedOperation = bombooTree.search(operation);
                insertedOperation.setType(NodeType.OPERATION);
            } else {
                BOO operation = new BOO(quantity, name);
                bombooTree.insert(operation);
                BOO insertedOperation = bombooTree.search(operation);
                insertedOperation.setType(NodeType.OPERATION);
                insertedOperation.setItems(firstElement.getItems());

insertedOperation.setQuantityItems(firstElement.getQuantityItems());
            }
        } else if (node.getType() == NodeType.MATERIAL) {
            if (bombooTree.getLatestInsertedNode() != null) {
                BOO latestOperation =
bombooTree.search(bombooTree.getElem(bombooTree.getLatestInsertedNode()));
                if (latestOperation != null) {
                    latestOperation.addItems(name);
                    latestOperation.addQuantity(quantity);
                }
            } else {
                firstElement.addItems(name);
                firstElement.addQuantity(quantity);
            }
        }
    }

    List<TreeNode<String>> materialChildren = new ArrayList<>();
    List<TreeNode<String>> operationChildren = new ArrayList<>();

    for (TreeNode<String> child : node.getChildren()) {
        if (child.getType() == NodeType.MATERIAL) {
            materialChildren.add(child);
        } else if (child.getType() == NodeType.OPERATION) {
            operationChildren.add(child);
        }
    }

    for (TreeNode<String> materialChild : materialChildren) {
        createBOMBOOTree(materialChild, firstElement);
```

```
    }

    for (TreeNode<String> operationChild : operationChildren) {
        createBOMBOOTree(operationChild, firstElement);
    }
}
```

- Time Complexity: O(n)
  - n = number of nodes in the tree
- Operations:
  - String processing: O(1) per node
  - Tree operations (insert/search): O(log h) where h is tree height
  - Child processing: O(c) where c is number of children per node
- Space Complexity: O(h) due to recursion stack, where h is tree height

```
private Iterable<BOO> changeOrder(List<BOO> postOrder) {
    List<BOO> postOrderElements = new ArrayList<>();
    for (int i = postOrder.size() - 1; i >= 0; i--) {
        postOrderElements.add(postOrder.get(i));
    }
    return postOrderElements;
}
```

- Time Complexity: O(n)
  - n = length of postOrder list
- Single pass through the list in reverse order
- Space Complexity: O(n) for new list creation

```
private void fillMaterials(LinkedHashMap<Integer, BOO> materials,
Iterable<BOO> postOrderElements) {
    int index = 0;
    for (BOO boo : postOrderElements) {
        materials.put(index++, boo);
    }
}
```

- Time Complexity: O(n)
  - n = number of elements in postOrderElements
- Single pass through the elements
- Space Complexity: O(n) in the materials map

```
private static void removeNullMachines(ArrayList<Workstation> workstations)
{
    workstations.removeIf(machine -> machine.getId().equalsIgnoreCase(""));
}
```

- Time Complexity: O(n)
  - n = number of workstations
- Single pass through the workstations list using removeIf()

```
private void fillItemsWithMaterials(LinkedHashMap<Integer, BOO> materials,
ArrayList<Material> filteredMaterials) {
    ItemsRepository itemsRepository =
Instances.getInstance().getItemsRepository();
    List<String> itemsIDs = new
```

```
ArrayList<>(itemsRepository.getItemsRepository().keySet());
    for (Map.Entry<Integer, BOO> entry : materials.entrySet()) {
        BOO boo = entry.getValue();
        List<String> materialNames = boo.getItems();
        for (String materialName : materialNames) {
            for (String id : itemsIDs) {
                if
(itemsRepository.getItemValue(id).equalsIgnoreCase(materialName)) {
                    Double quantity =
boo.getQuantityItems().get(boo.getItemPosition(materialName)); // Get the
quantity of the material
                    String quantityString = String.valueOf(quantity);
                    Material material = new Material(id, materialName,
quantityString);
                    filteredMaterials.add(material);
                    break;
                }
            }
        }
    }
}
```

- Time Complexity: O(n * m * k)
  - n = number of materials
  - m = number of material names
  - k = number of item IDs
- Nested loops through materials, names, and IDs

```
private static void fillOperationsQueue(ArrayList<Material> materials,
LinkedHashMap<String, LinkedList<Material>> operationsQueue, List<BOO>
postOrderElements) {
    for (BOO boo : postOrderElements) {
        String operation = boo.getOperation();
        for (String itemName : boo.getItems()) {
            for (Material material : materials) {
                if (material.getName().equalsIgnoreCase(itemName)) {
                    if (!operationsQueue.containsKey(operation)) {
                        operationsQueue.put(operation, new LinkedList<>());
                    }
                    operationsQueue.get(operation).add(material);
                }
            }
        }
    }
}
```

- Time Complexity: O(n * m * k)
  - n = number of BOO elements
  - m = number of items per BOO
  - k = number of materials
- Three nested loops iterating through BOOs, items, and materials

```
private void fillUpMachinesUS16(LinkedHashMap<String, LinkedList<Material>>
operationsQueue, ArrayList<Workstation> workstations, ArrayList<Material>
materials) {
    int quantMachines = workstations.size();
    sortMachinesByTime(workstations);
    addAllItemsWithSteps(operationsQueue, workstations, materials,
```

```
quantMachines);
}
```

- Time Complexity: O(n log n + m * k)
  - n = number of workstations (for sorting)
  - m * k = complexity from addAllItemsWithSteps
- Includes sorting workstations (O(n log n)) and processing all items

```
private static void sortMachinesByTime(ArrayList<Workstation> workstations)
{
    workstations.sort(Comparator.comparingInt(Workstation::getTime));
}
```

- Time Complexity: O(n log n)
  - n = number of workstations
- Uses Java's built-in sorting algorithm

```
private void addAllItemsWithSteps(HashMap<String, LinkedList<Material>>
operationsQueue, ArrayList<Workstation> workstations, ArrayList<Material>
materials, int quantMachines) {
    for (String operation : operationsQueue.keySet()) {
        for (Material material : materials) {
            if (!material.getID().equalsIgnoreCase("") &&
operationsQueue.get(operation).contains(material)) {
                quantMachines = addOperationsWithSteps(operation,
workstations, material, quantMachines, operationsQueue);
            }
        }
    }
}
```

- Time Complexity: O(n * m * k)
  - n = number of operations
  - m = number of materials
  - k = number of workstations
- Nested loops through operations, materials, and workstation processing

```
private int addOperationsWithSteps(String operation, ArrayList<Workstation>
workstations, Material material, int quantMachines, HashMap<String,
LinkedList<Material>> operationsQueue) {
    ArrayList<Workstation> availableWorkstations = new ArrayList<>();
    for (Workstation workstation : workstations) {
        if (workstation.getOperationName().equalsIgnoreCase(operation)) {
            availableWorkstations.add(workstation);
        }
    }
    if (quantMachines == 0) {
        quantMachines = checkMachines(workstations, quantMachines);
    } else {
        quantMachines = checkMachinesWithOperation(availableWorkstations,
quantMachines, operation);
    }
    quantMachines = addItemsWithSteps(operationsQueue, material, operation,
availableWorkstations, quantMachines);
    return quantMachines;
}
```

- Time Complexity: O(n)
    - n = number of workstations
- Linear scan through workstations to find available ones

```
private static int checkMachinesWithOperation(ArrayList<Workstation>
workstations, int quantMachines, String operation) {
    boolean notFree = true;
    int quant = 0;
    ArrayList<Workstation> tempMachines = new ArrayList<>();
    for (Workstation workstation : workstations) {
        if (workstation.getOperationName().contains(operation) &&
workstation.getHasItem()) {
            quant++;
            tempMachines.add(workstation);
        }
        if (workstation.getOperationName().equalsIgnoreCase(operation) &&
!workstation.getHasItem()) {
            notFree = false;
        }
    }
    if (quant >= 1 && notFree) {
        for (Workstation workstation : tempMachines) {
            workstation.clearUpWorkstation();
        }
        quantMachines = quantMachines + quant;
    }
    return quantMachines;
}
```

- Time Complexity: O(n)
    - n = number of workstations
- Linear scan through workstations twice

```
private static int checkMachines(ArrayList<Workstation> machines, int
quantMachines) {
    if (quantMachines == 0) {
        for (Workstation machine1 : machines) {
            machine1.clearUpWorkstation();
        }
        quantMachines = machines.size();
    }
    return quantMachines;
}
```

- Time Complexity: O(n)
    - n = number of workstations
- Single pass through machines list

```
private int addItemsWithSteps(HashMap<String, LinkedList<Material>>
operationsQueue, Material material, String operation,
ArrayList<Workstation> availableWorkstations, int quantMachines) {
    for (Workstation workstation : availableWorkstations) {
        if
((operationsQueue.get(workstation.getOperation().getDescription()).contains
(material) && workstation.getOperationName().equalsIgnoreCase(operation))
&& (!workstation.getHasItem())) {
            int currentItem = timeOperations.size() + 1;
            workstation.setHasItem(true);
```

```
            quantMachines--;
            String operation1 = currentItem + " - " + "Operation: " +
operation + " - Machine: " + workstation.getId() + " - Item: " +
material.getName() + " - Time: " + workstation.getTime() + " - Quantity: "
+ material.getQuantity();
            timeOperations.put(operation1, (double) workstation.getTime());

operationsQueue.get(workstation.getOperation().getDescription()).remove(mat
erial);
            return quantMachines;
        }
    }
    return quantMachines;
}
```

- Time Complexity: O(n)
  - n = number of available workstations
- Linear scan through available workstations

```
private LinkedHashMap<String, Double>
SimulatorResetUS16(ArrayList<Workstation> workstations, ArrayList<Material>
filteredMaterials) {
    for (Workstation workstation : workstations) {
        workstation.clearUpWorkstation();
    }
    return timeOperations;
}
```

- Time Complexity: O(n)
  - n = number of workstations
- Single pass through workstations to clear them
- Returns existing timeOperations map without additional processing
- Space Complexity: O(1) - no additional space needed beyond input

Overall Complexity

The program's total complexity remains O(n * m * k) due to the nested operations in the data processing phase, despite the final reset phase being only O(n).