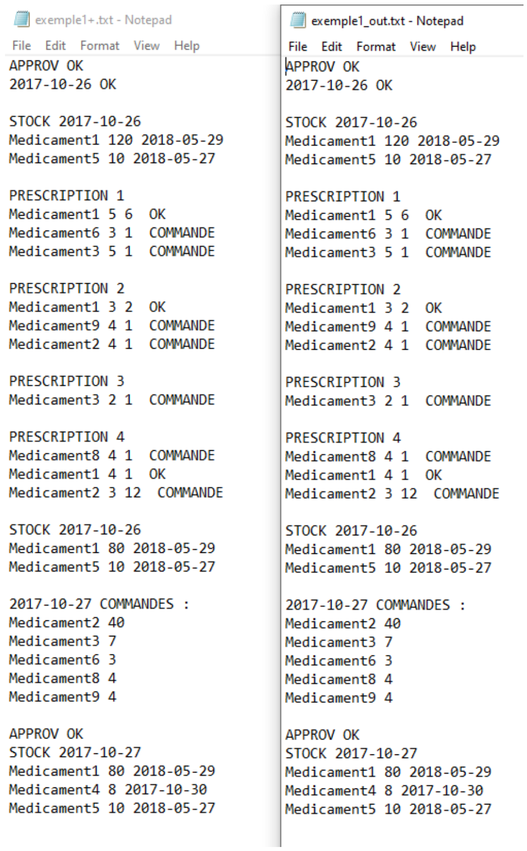
IFT2015 ASSIGNMENT 2

July 9, 2023

Massimo Pietracupa (20206087) and Vincent Hoang (20183549)

1: Auto-evaluation: Specify if your program works correctly, partially or does not work at all.

Our program seems to be working correctly. When we compare the outputs that are received to those that were provided by the lab instructor, the outputs matched exactly:


exemple1+.txt - Notepad
File Edit Format View Help
exemple1_out.txt - Notepad
File Edit Format View Help

```
APPROV OK
2017-10-26 OK

STOCK 2017-10-26
Medicament1 120 2018-05-29
Medicament5 10 2018-05-27

PRESCRIPTION 1
Medicament1 5 6  OK
Medicament6 3 1  COMMANDE
Medicament3 5 1  COMMANDE

PRESCRIPTION 2
Medicament1 3 2  OK
Medicament9 4 1  COMMANDE
Medicament2 4 1  COMMANDE

PRESCRIPTION 3
Medicament3 2 1  COMMANDE

PRESCRIPTION 4
Medicament8 4 1  COMMANDE
Medicament1 4 1  OK
Medicament2 3 12  COMMANDE

STOCK 2017-10-26
Medicament1 80 2018-05-29
Medicament5 10 2018-05-27

2017-10-27 COMMANDES :
Medicament2 40
Medicament3 7
Medicament6 3
Medicament8 4
Medicament9 4

APPROV OK
STOCK 2017-10-27
Medicament1 80 2018-05-29
Medicament4 8 2017-10-30
Medicament5 10 2018-05-27
```

```
APPROV OK
2017-10-26 OK

STOCK 2017-10-26
Medicament1 120 2018-05-29
Medicament5 10 2018-05-27

PRESCRIPTION 1
Medicament1 5 6  OK
Medicament6 3 1  COMMANDE
Medicament3 5 1  COMMANDE

PRESCRIPTION 2
Medicament1 3 2  OK
Medicament9 4 1  COMMANDE
Medicament2 4 1  COMMANDE

PRESCRIPTION 3
Medicament3 2 1  COMMANDE

PRESCRIPTION 4
Medicament8 4 1  COMMANDE
Medicament1 4 1  OK
Medicament2 3 12  COMMANDE

STOCK 2017-10-26
Medicament1 80 2018-05-29
Medicament5 10 2018-05-27

2017-10-27 COMMANDES :
Medicament2 40
Medicament3 7
Medicament6 3
Medicament8 4
Medicament9 4

APPROV OK
STOCK 2017-10-27
Medicament1 80 2018-05-29
Medicament4 8 2017-10-30
Medicament5 10 2018-05-27
```

The same is true for all other example files.

2: Time complexity analysis in Big O notation of the following transactions:

PRESCRIPTION;

APPROV;

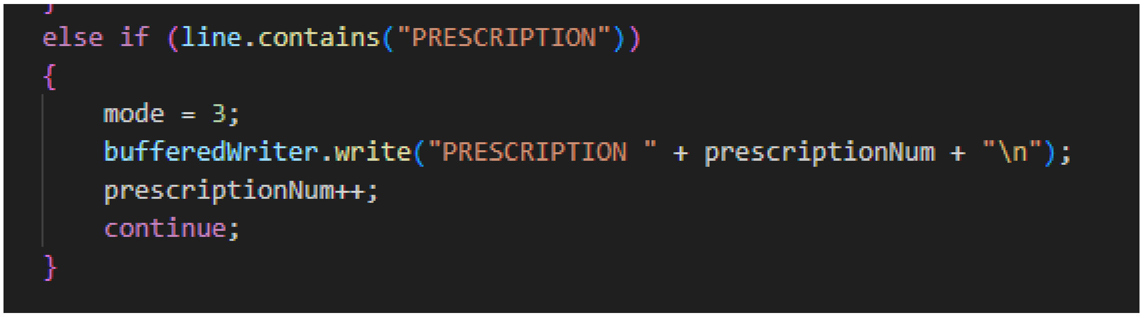
DATE;

Algorithm running time should be expressed in terms of:

- **n – number of different medications.**
- **m – number of items on a prescription.**
- **k – number of items on the order list (COMMANDES)**
- **and any other variable that you consider would be necessary.**

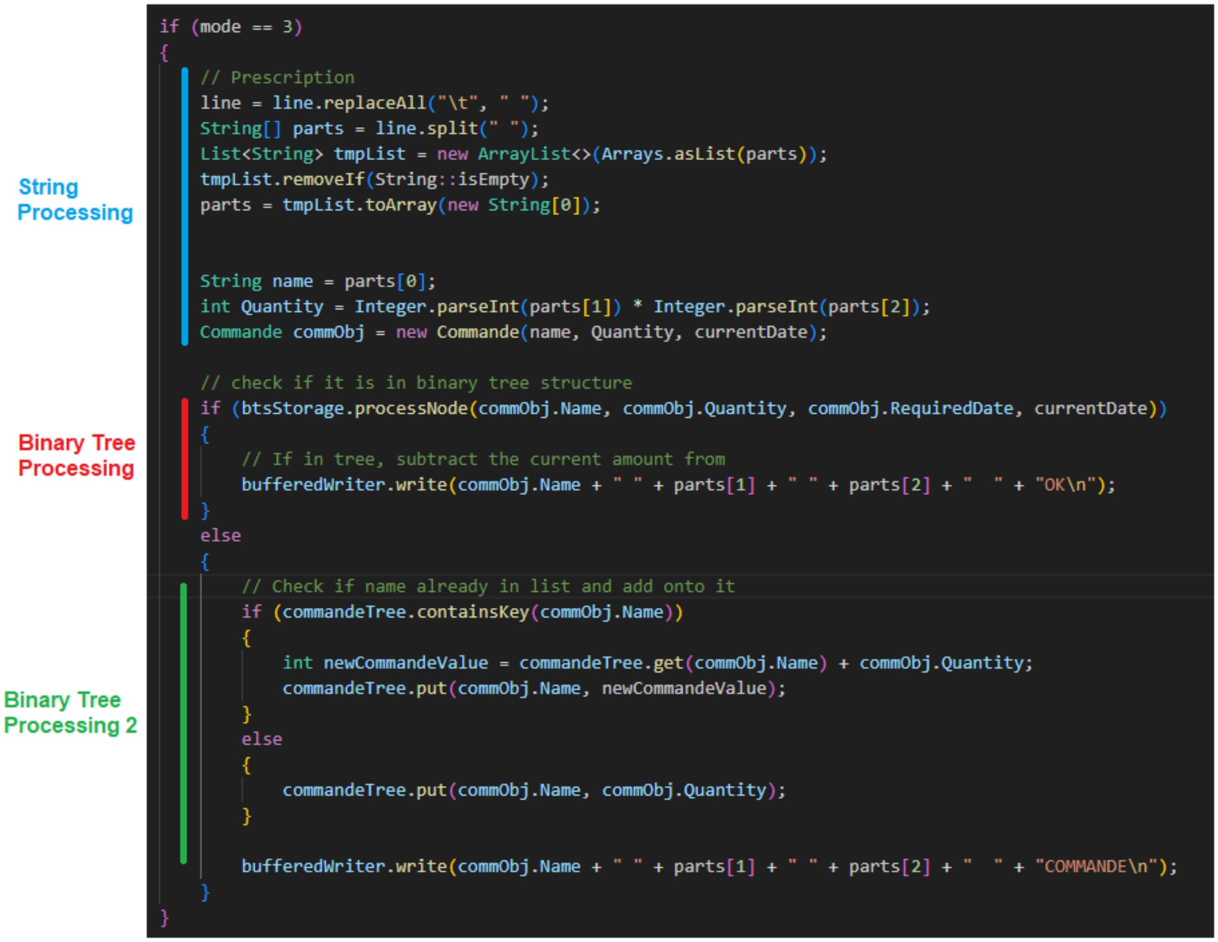
PRESCRIPTION:

When a transaction is identified as a prescription, the code first changes its mode to mode 3, to indicate that it should be processing prescriptions, as seen in the code below:



```
}
else if (line.contains("PRESCRIPTION"))
{
    mode = 3;
    bufferedWriter.write("PRESCRIPTION " + prescriptionNum + "\n");
    prescriptionNum++;
    continue;
}
```

When a new line prescription is found, it is first processed to remove tabs, then split into its 3 main components: Medication Name, quantity, and frequency. From here, we can now process the request using the binary tree structure. If it is not inside the binary tree, the prescription is stored inside a secondary binary tree using the java TreeMap class.

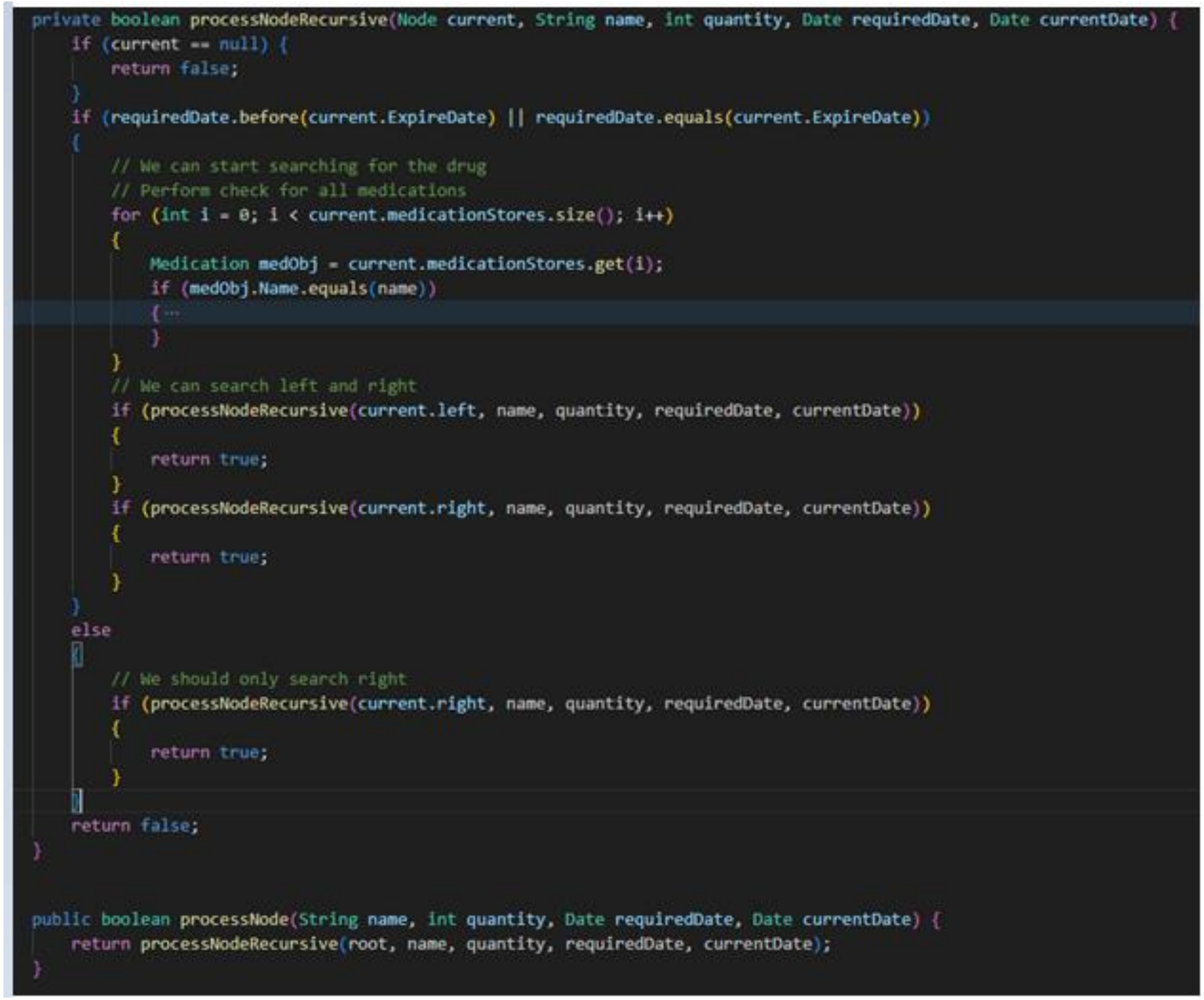


```
if (mode == 3)
{
    // Prescription
    line = line.replaceAll("\t", " ");
    String[] parts = line.split(" ");
    List<String> tmpList = new ArrayList<>(Arrays.asList(parts));
    tmpList.removeIf(String::isEmpty);
    parts = tmpList.toArray(new String[0]);


    String name = parts[0];
    int Quantity = Integer.parseInt(parts[1]) * Integer.parseInt(parts[2]);
    Commande commObj = new Commande(name, Quantity, currentDate);

    // check if it is in binary tree structure
    if (btsStorage.processNode(commObj.Name, commObj.Quantity, commObj.RequiredDate, currentDate))
    {
        // If in tree, subtract the current amount from
        bufferedWriter.write(commObj.Name + " " + parts[1] + " " + parts[2] + "  " + "OK\n");
    }
    else
    {
        // Check if name already in list and add onto it
        if (commandeTree.containsKey(commObj.Name))
        {
            int newCommandeValue = commandeTree.get(commObj.Name) + commObj.Quantity;
            commandeTree.put(commObj.Name, newCommandeValue);
        }
        else
        {
            commandeTree.put(commObj.Name, commObj.Quantity);
        }

        bufferedWriter.write(commObj.Name + " " + parts[1] + " " + parts[2] + "  " + "COMMANDE\n");
    }
}
```

The time complexity of the string processing will not vary with respect to the number of medications, prescriptions, or items on the order list, and will remain constant with regards to the input string, which should remain constant. Hence the time complexity for this segment will remain O(1).

We will first examine the TreeMap operations. The time complexity for the containsKey operation for TreeMaps are those of binary trees, typically O(logm), as well as the "put" functions, hence the overall time complexity for the hash map processing will remain O(logm), where m is the unique number of prescriptions.

In order to examine the time complexity of the binary tree processing, we need to examine the processNode function of the binary tree, as seen below.



```java
private boolean processNodeRecursive(Node current, String name, int quantity, Date requiredDate, Date currentDate) {
    if (current == null) {
        return false;
    }
    if (requiredDate.before(current.ExpireDate) || requiredDate.equals(current.ExpireDate))
    {
        // We can start searching for the drug
        // Perform check for all medications
        for (int i = 0; i < current.medicationStores.size(); i++)
        {
            Medication medObj = current.medicationStores.get(i);
            if (medObj.Name.equals(name))
            {…
            }
        }
        // We can search left and right
        if (processNodeRecursive(current.left, name, quantity, requiredDate, currentDate))
        {
            return true;
        }
        if (processNodeRecursive(current.right, name, quantity, requiredDate, currentDate))
        {
            return true;
        }
    }
    else
    {
        // We should only search right
        if (processNodeRecursive(current.right, name, quantity, requiredDate, currentDate))
        {
            return true;
        }
    }
    return false;
}


public boolean processNode(String name, int quantity, Date requiredDate, Date currentDate) {
    return processNodeRecursive(root, name, quantity, requiredDate, currentDate);
}
```

A recursive function, processNodeRecursive, is called that essentially uses the current date to parse for valid drugs within the tree. Nodes are stored using their expire date, which allows for effective range searches within the tree. If the required date, which is calculated using the prescription information is before the expire date of node, we can recursively check left or right down the tree. If it is not, then we simply search right, which can potentially cut down the search significantly. If more than one drug contain the same expire date, the medication will be appended onto the node's drug array. Once a medication's name is found within the acceptable date range, it is processed and subtracted from the storage total and a true value is returned, otherwise false is returned. Hence, in order to examine the time complexity of this function, we need to consider two factors:

1. Binary tree structure
2. Number of medications within a node list

If we examine the first point, and consider the number of different type of expire dates as "p" the binary tree structure will have a time complexity of O(logp), provided that the tree is balanced. It is important to note that for this particular binary tree, AVL tree structure is considered and the tree is rebalanced based on the imbalance factor each time a node is added. With this information, we know that we can achieve a time complexity of O(logp) when we decide to parse the tree for an average case. For a case where all the medication have the exact same expire date, then the time complexity would essentially be O(n), where n represents the number of different medications. So if we combine these, our total time complexity would be O(logpn).

If we combine all these components, we have a total time complexity of:

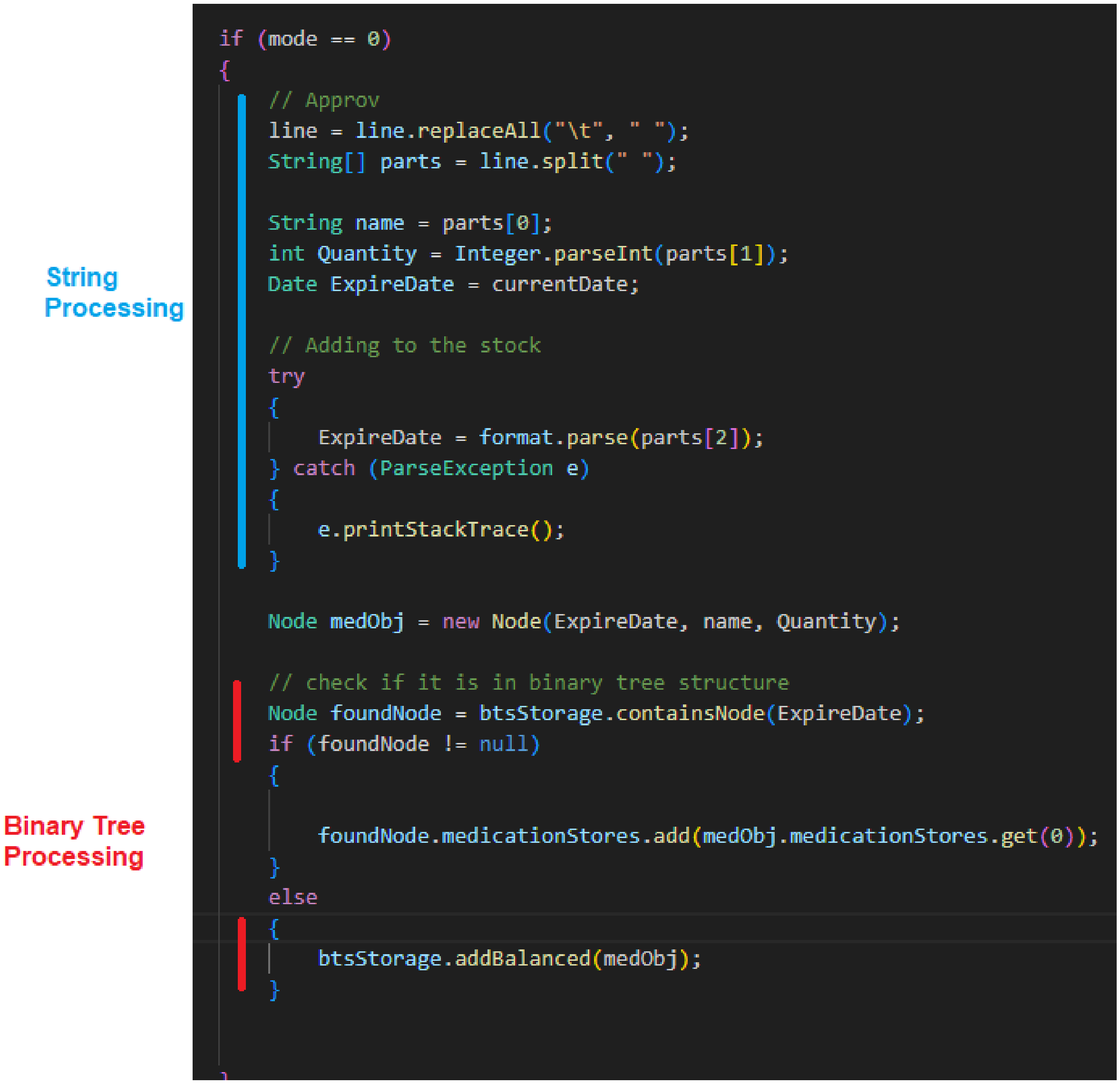
O(1) + O(logpn) + O(logm) = **O(logpn + logm)**

Continued:

We can essentially achieve an even higher efficiency if we create another balanced binary tree structure, according to the medication number, within each node. Again, it would have a time complexity of O(logn) meaning that the overall time complexity would be O(logp*logn), which is more efficient than our current implementation of O(logpn). Again architecture is designed based on need and based on the examples provided, it did not seem likely that a scenario where all medications would contain the same expire date, instead all containing unique dates. This means that the programs added maintenance complexity of double binary trees did not seem necessary as based on previous data, the complexity is already trending closer to O(logp) with the current implementation.

APPROV :

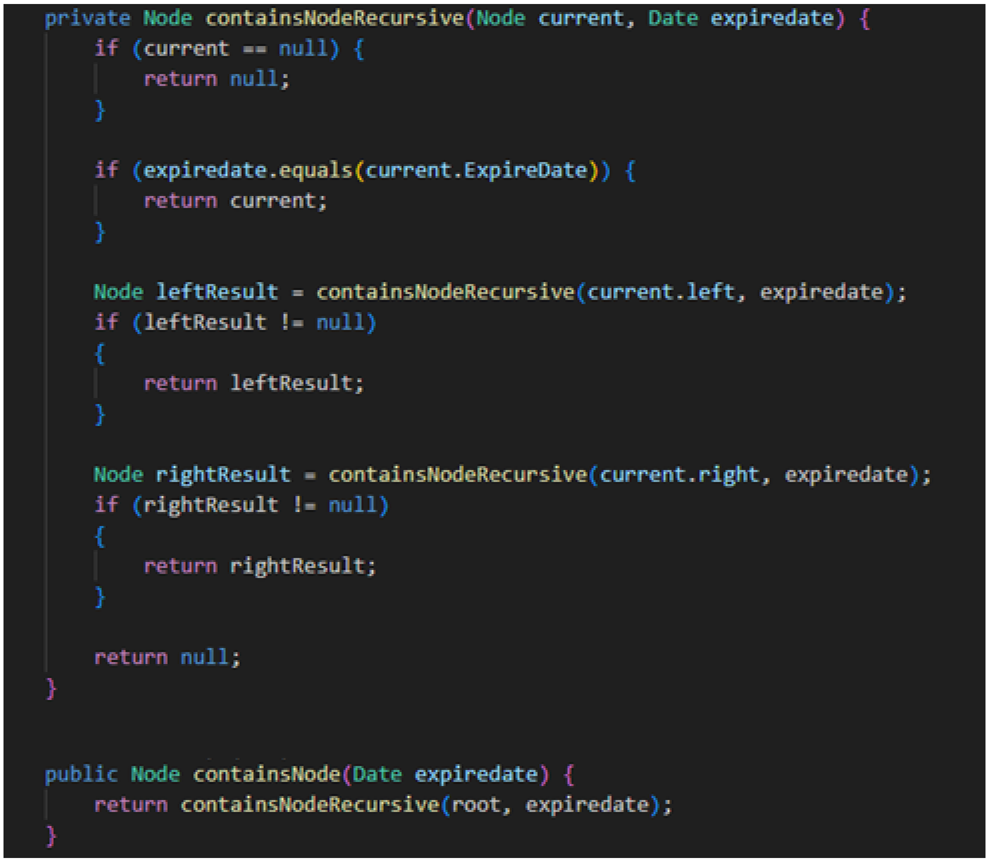
For the Approv sequence, the code enables mode 0. There are two components to the mode 0, the string processing and the binary tree processing.



```java
if (mode == 0)
{
    // Approv
    line = line.replaceAll("\t", " ");
    String[] parts = line.split(" ");

    String name = parts[0];
    int Quantity = Integer.parseInt(parts[1]);
    Date ExpireDate = currentDate;

    // Adding to the stock
    try
    {
        ExpireDate = format.parse(parts[2]);
    } catch (ParseException e)
    {
        e.printStackTrace();
    }

    Node medObj = new Node(ExpireDate, name, Quantity);

    // check if it is in binary tree structure
    Node foundNode = btsStorage.containsNode(ExpireDate);
    if (foundNode != null)
    {

        foundNode.medicationStores.add(medObj.medicationStores.get(0));
    }
    else
    {
        btsStorage.addBalanced(medObj);
    }


}
```

Once again, the string processing is independent of any variables in the program. It will be at a time complexity of O(1).

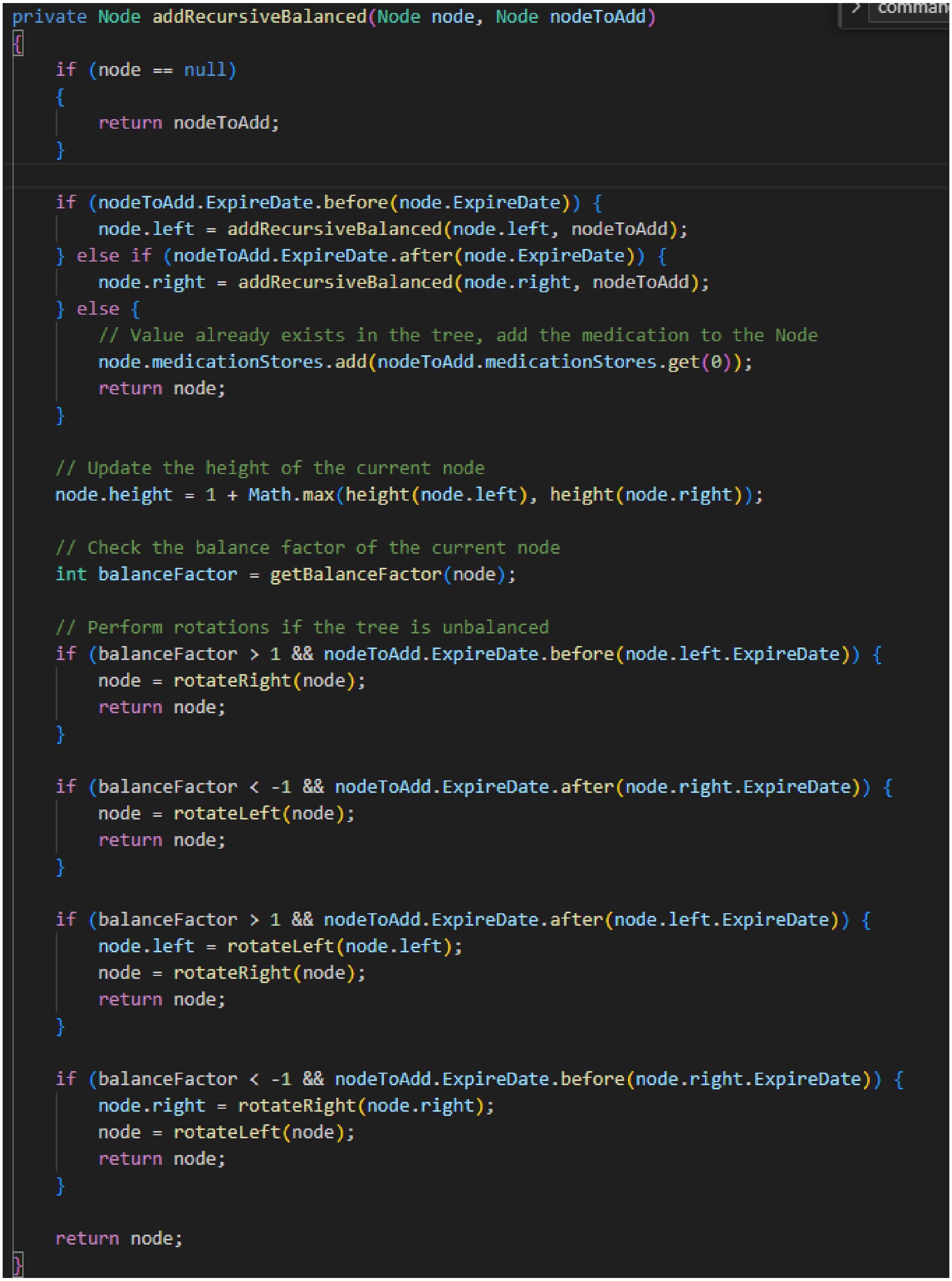
The containsNode function of the binary tree is quite simple and can be seen below:



```java
private Node containsNodeRecursive(Node current, Date expiredate) {
    if (current == null) {
        return null;
    }

    if (expiredate.equals(current.ExpireDate)) {
        return current;
    }

    Node leftResult = containsNodeRecursive(current.left, expiredate);
    if (leftResult != null)
    {
        return leftResult;
    }

    Node rightResult = containsNodeRecursive(current.right, expiredate);
    if (rightResult != null)
    {
        return rightResult;
    }

    return null;
}


public Node containsNode(Date expiredate) {
    return containsNodeRecursive(root, expiredate);
}
```

It simply navigates through the binary tree to find a particular node containing the expire date. If one is found, that means that we simply need to append onto the medications list that it contains. If not, then a new node would need to be added. This function is performed at an average O(logp) time, assuming the tree is balanced, which it is due to the addbalanced function.

The addbalanced function is a bit more complex and calls the addRecursiveBalanced function, as seen below. This function adds a node to the tree, then calculates the balance factor of the current node and performs rotations as needed to rebalance the tree. These rotations are the same as AVL trees. Rotations are performed in O(1) time. The getBalanceFactor function also has a time complexity of O(1). Seeing as these functions can be ignored, the binary tree search itself will take an average O(logp) time, where p is the number of unique expire dates.



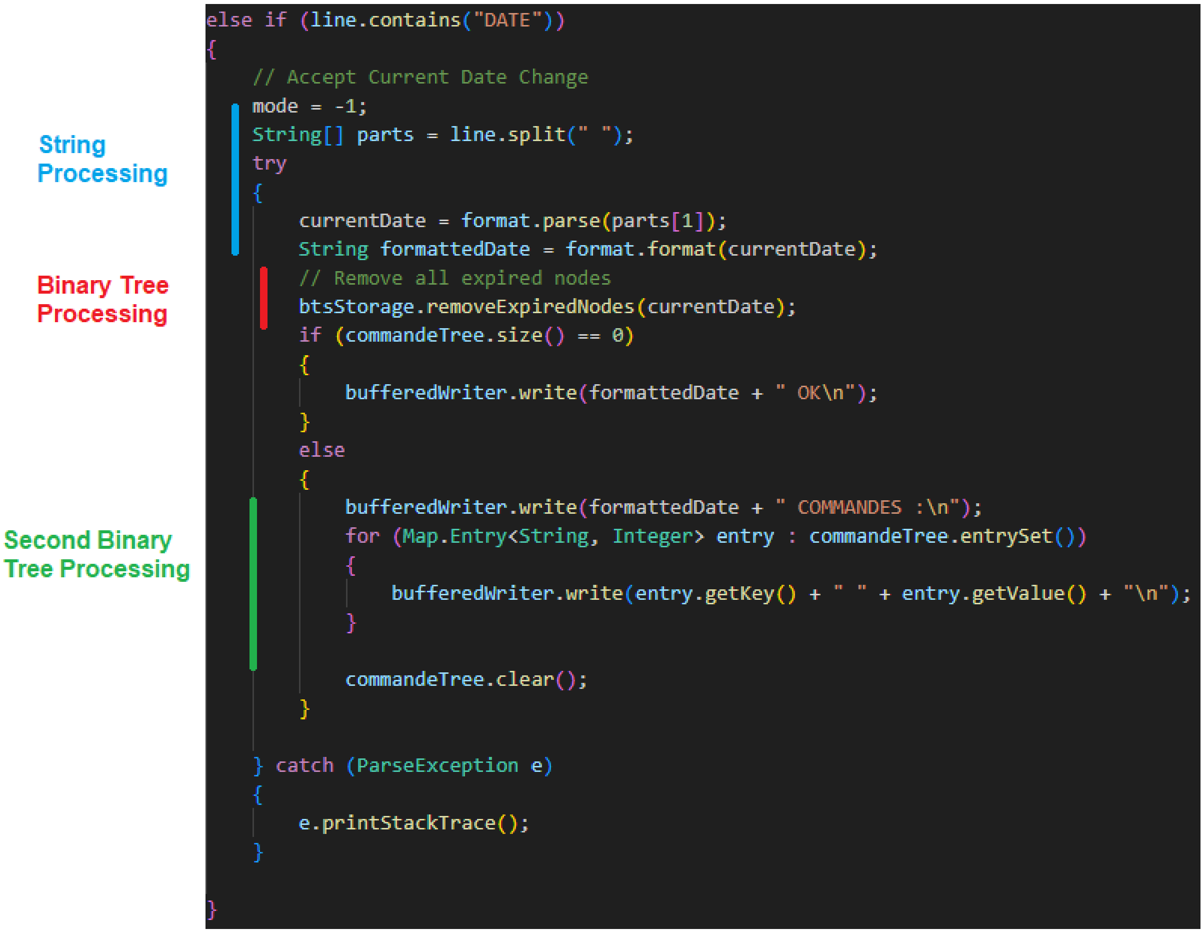
```java
private Node addRecursiveBalanced(Node node, Node nodeToAdd)
{
    if (node == null)
    {
        return nodeToAdd;
    }

    if (nodeToAdd.ExpireDate.before(node.ExpireDate)) {
        node.left = addRecursiveBalanced(node.left, nodeToAdd);
    } else if (nodeToAdd.ExpireDate.after(node.ExpireDate)) {
        node.right = addRecursiveBalanced(node.right, nodeToAdd);
    } else {
        // Value already exists in the tree, add the medication to the Node
        node.medicationStores.add(nodeToAdd.medicationStores.get(0));
        return node;
    }

    // Update the height of the current node
    node.height = 1 + Math.max(height(node.left), height(node.right));

    // Check the balance factor of the current node
    int balanceFactor = getBalanceFactor(node);

    // Perform rotations if the tree is unbalanced
    if (balanceFactor > 1 && nodeToAdd.ExpireDate.before(node.left.ExpireDate)) {
        node = rotateRight(node);
        return node;
    }

    if (balanceFactor < -1 && nodeToAdd.ExpireDate.after(node.right.ExpireDate)) {
        node = rotateLeft(node);
        return node;
    }

    if (balanceFactor > 1 && nodeToAdd.ExpireDate.after(node.left.ExpireDate)) {
        node.left = rotateLeft(node.left);
        node = rotateRight(node);
        return node;
    }

    if (balanceFactor < -1 && nodeToAdd.ExpireDate.before(node.right.ExpireDate)) {
        node.right = rotateRight(node.right);
        node = rotateLeft(node);
        return node;
    }

    return node;
}
```

Overall if we add up the times, we receive the following:

O(1) + O(logp) + O(logp) = **O(logp)**

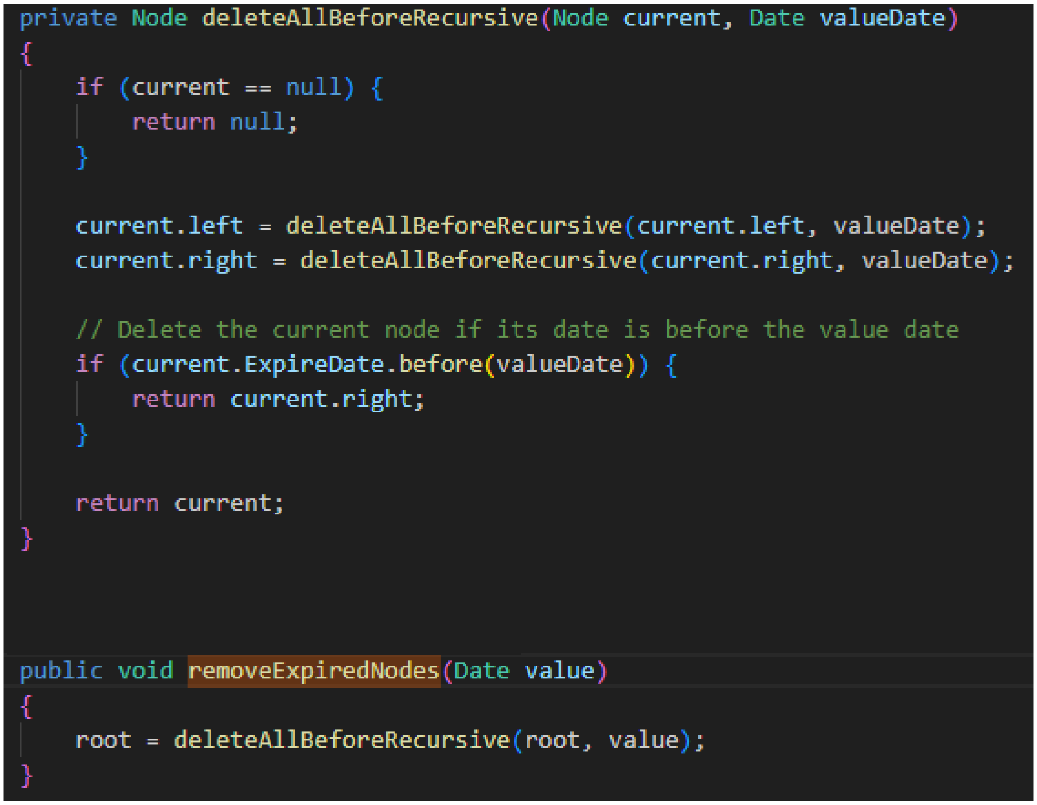
DATE :

The date section performs some string processing, which has a time complexity of O(1). Once that is completed, the Binary Tree is cleaned up in accordance to the updated date, meaning all nodes that have an expire date before the current date are purged.



```java
else if (line.contains("DATE"))
{
    // Accept Current Date Change
    mode = -1;
    String[] parts = line.split(" ");
    try
    {
        currentDate = format.parse(parts[1]);
        String formattedDate = format.format(currentDate);
        // Remove all expired nodes
        btsStorage.removeExpiredNodes(currentDate);
        if (commandeTree.size() == 0)
        {
            bufferedWriter.write(formattedDate + " OK\n");
        }
        else
        {
            bufferedWriter.write(formattedDate + " COMMANDES :\n");
            for (Map.Entry<String, Integer> entry : commandeTree.entrySet())
            {
                bufferedWriter.write(entry.getKey() + " " + entry.getValue() + "\n");
            }

            commandeTree.clear();
        }

    } catch (ParseException e)
    {
        e.printStackTrace();
    }

}
```

The removeExpiredNodes function can be seen below, which utilizes the power of the range search in binary trees and leveraging the expire date key of the nodes to quickly purge nodes. At worst case, the time complexity would be O(logp), where p represents the unique number of expire dates.



```
private Node deleteAllBeforeRecursive(Node current, Date valueDate)
{
    if (current == null) {
        return null;
    }

    current.left = deleteAllBeforeRecursive(current.left, valueDate);
    current.right = deleteAllBeforeRecursive(current.right, valueDate);

    // Delete the current node if its date is before the value date
    if (current.ExpireDate.before(valueDate)) {
        return current.right;
    }

    return current;
}



public void removeExpiredNodes(Date value)
{
    root = deleteAllBeforeRecursive(root, value);
}
```

Finally, cycling through the TreeMap has a time complexity of O(k), where k represents number of orders on the order list. The clear operation should take O(1) time.

Overall, we can see that the overall time complexity of the Date operation is as follows:

O(1) + O(logp) + O(k) = **O(logp + k)**