Report

Mamadou Diao Kaba (27070179)

Lifu Zhang (40081513)

Problem 1.1

```
public InventoryRecord[] createListIndex(String attribute)
{
    final InventoryRecord[] listCopy = copyList();
    util.QuickSort(listCopy, attribute);
    //util.print(listCopy); // Print the sorted list
    ArrayIndex=listCopy;
    return listCopy;
}

// Create an array containing every record in the dictionnary
InventoryRecord[] copyList()
{
    // Create array of size of the list
    InventoryRecord[] listCopy = new InventoryRecord[vList.size()];
    vList.moveToStart(); // Move at the beginning of the list
    while(vList.currPos() < vList.size()) { // Iterate through the list
        listCopy[vList.currPos()] = vList.getValue(); // Copy the list content in the array
        vList.next(); // Check next record
    }
    return listCopy;
}
```

The time complexity of the **createListIndex** method is determined by the time complexity of two main operations:

1. The Method **copyList** method, which has a time complexity of **O(n)**, where "n" is the number of elements in the dictionary.
2. Sorting the copied array using util.QuickSort: The time complexity of the **QuickSort** algorithm in the average and best-case time complexity is **O(nlog(n))**

So, the dominant factor in the time complexity of the **createListIndex** method is the sorting step, which is **O(nlog(n))** on average and in the best case.

Problem 1.2

```java
public void insert(double val,InventoryRecord record)
{
    root = inserthelp(root, val,record);
    nodecount++;
}

private BinarySearchTreeNode inserthelp(BinarySearchTreeNode root,double val,InventoryRecord record)
{


    if (root == null) return new BinarySearchTreeNode(val,record);
    if (val<root.getRoot())
        root.setLeft(inserthelp(root.left(),val, record));
    else
        root.setRight(inserthelp(root.right(), val,record));
    return root;

}
```

The **insert** method has a time complexity **of O(log(n))** on average, where "n" is the number of elements in the binary search tree.

```java
public BinarySearchTree createTreeIndex(String attribute)
{
    BinarySearchTree BST=new BinarySearchTree();
    switch(attribute)
    {
    case "unitprice":

        this.vList.moveToStart();

        for(int i=0;i<this.vList.size();i++)
        {
            BST.insert(this.vList.getValue().getUnitPrice(),vList.getValue());
            this.vList.next();
        }
        break;
    case "quantityinstock":
        for(int i=0;i<this.getSize();i++)
        {
            BST.insert(vList.getValue().getQtyInStock(),vList.getValue());
            vList.next();
        }
        break;
    case "inventroyvalue":
        for(int i=0;i<this.getSize();i++)
        {
            BST.insert(vList.getValue().getInventoryValue(),vList.getValue());
            vList.next();
        }
        break;
    case "recorderlevel":
        for(int i=0;i<this.getSize();i++)
        {
            BST.insert(vList.getValue().getReorderLevel(),vList.getValue());
            vList.next();
        }
        break;
    case "reordertime":
        for(int i=0;i<this.getSize();i++)
        {
            BST.insert(vList.getValue().getReorderTime(),vList.getValue());
            vList.next();
        }
        break;
```

```
    case "reorderqty":
        for(int i=0;i<this.getSize();i++)
        {
            BST.insert(vList.getValue().getReorderQty(),vList.getValue());
            vList.next();
        }
        break;
    }
    BSTIndex= BST;
    return  BST;
}
```

The **createTreeIndex** method, which creates a binary search tree index for a specific attribute, has a time complexity of O(nlog(n)), where "n" is the number of elements in the dictionary.

The insertion operation takes O(log(n)) time on average, and since we do this for all "n" elements, the overall time complexity is **O(nlog(n)).**