Assignment II: Supermarket statistics

Jip Derksen (500839909)
Yaël de Vries (500849315)
IS202-5

Introduction

After our first assignment we have received feedback which made us able to understand the assignment better. We have gotten a clearer and broader idea of what needs to happen to pass the assignments, whether it is about the code or the documentation. In this document we will we getting into why we made certain choices within our code by using code snippets and an explanations provided with these snippets. We will be explaining about certain elements take part in our code and why we chose to make use of these elements. After last assignment our class has gotten a lot of feedback regarding the documentation and what was done well as well as what we could improve in our way of documentation. We will be trying to implement our individual feedback as well as the collective feedback provided in the lectures during this assignment and are optimistic about the outcome.

Now about the assignment. During the classes that we have had when creating the second assignment for ADS we have come across a multitude of subjects to learn. One of these subjects is how fast different algorithms are and the use case of all those different methods. Another was lambda expressions, how to give a function as a parameter. And yet another one was about recursion. And recursion is something we have been using in the previous assignment as well, but we didn't follow any classes regarding recursion. For the second assignment we did get some exercises to practice our recursion, which already seemed to be quite the challenge. Nevertheless we managed as well as getting the assignment done in time. Together with all the work that we need to be doing for school it seems like a lot. And it is, but it is still doable.

In this document we will be focusing yet again on a minimum of seven code snippets with an explanation of why we made the choices that we did. We will be discussing subjects like the OrderedArrayList and the implementation of other methods that make use of the right representation invariant. Mostly we will be talking about the class orderedArrayList, as that is where most of the magic happens that we created.

This assignment contained some brain breakers of tests for us to solve and in our experience this required a lot more thinking than the first assignments about trains. Yet again we kept seeing errors rising when creating the code to complete the tests. Something we noticed is that some errors were as small as an expectation of *61* and actually gotten *62*. Some were even smaller when the expectation was *1* and actually gotten *1*. We got so confused for a while but we did managed to solve it.

Table of contents

Introduction 1
Table of contents 2
1. Requirements 3
Documentational requirements 3
Code requirements and assignment 3
2. add(int index, E element); 4
3. nSorted = 0; 5
4. remove(int index) & remove(Object o); 5
5. indexOf(Object item) & indexOfByBinarySearch(E searchItem); 6
6. indexOfByIterativeBinarySearch(); 7
7. indexOfByRecursiveBinarySearch(); 8
8. recursiveBinarySearch(); 9
9. TestObjectsExistence(); 10

1. Requirements

For this assignment there are requirements as well. For the code as well as for the document.
Below we will display the requirements and the assignment.

Documentational requirements

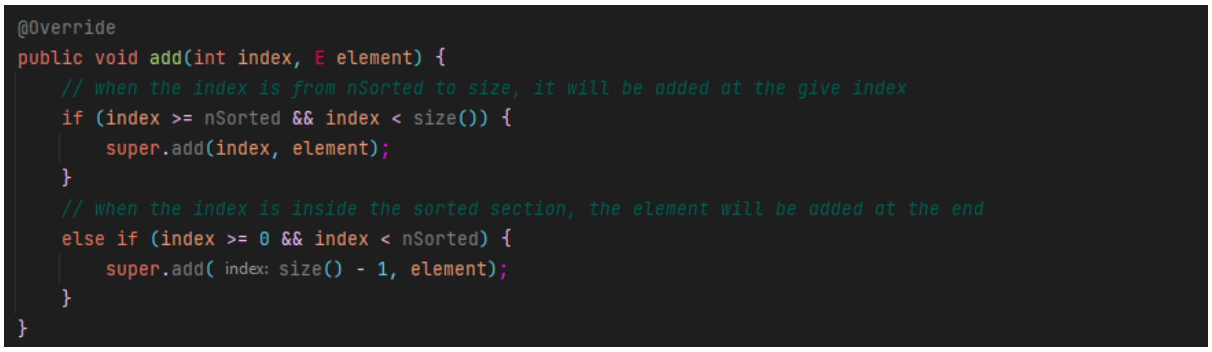
Prepare your document with explanations of seven code snippets.
Explain how the OrderedArrayList representation invariant is sustained by your overridden implementation of add(index,item) or one of the remove methods. (Choose the most relevant.)
Explain correctness of one code snippet by involving a loop invariant.

Then we had feedback from the previous assignment in which it was clear we had to explain the why instead of the what. Take a tour around our brains to see why we made our choices.

Code requirements and assignment

1. It is not allowed to change any of the signatures or specifications of public methods and classes that are provided in the starter project. You may add additional public and private methods as you wish.
2. Your solution should reproduce the sample output that is given in the case description to the extent that this output can be explained to follow directly from the given test data independent of your solution approach. Any mismatch should be explained in your report.
3. Your solution should pass every unit test provided, and you are expected to add more unit tests for complete coverage of your solution
4. Complete the implementations of the Product and Purchase classes:
   a) you need to complete the toString method to obtain a proper text representation in output.
   b) you need to complete the static fromLine methods which you will need for converting text lines of the source files into object instances.
5. Complete the implementation of the generic OrderedArrayList class, which implements the OrderedList interface. (If you find it hard to directly code the generic implementations of your algorithms you may also copy the template into a ProductsList class implementing OrderedList<Product>and replace all E's by Product. Similarly, you may duplicate into a PurchasesList class replacing all E's by Purchase).
   (Correct implementations of the overridden remove methods are optional).
6. Implement two versions of the binary search: an iterative version and a recursive version
7. Complete the implementation of the default method aggregate in the OrderedList<E> interface.
8. Implement the missing parts of the PurchaseTracker.
9. You may add more private and public methods to any of the classes and interfaces, but not change any signature of the specified public methods.
10. Run all provided unit tests and add unit tests for relevant code or situations that are not covered by the provided unit tests.

2. add(int index, E element);

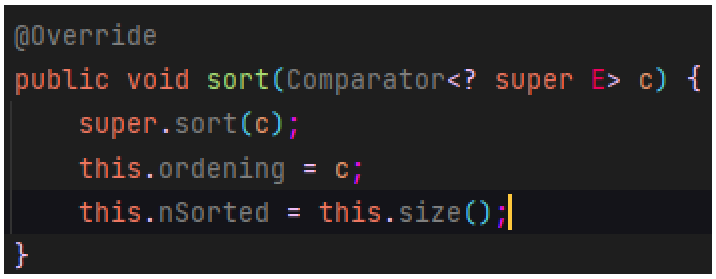


```java
@Override
public void add(int index, E element) {
    // when the index is from nSorted to size, it will be added at the give index
    if (index >= nSorted && index < size()) {
        super.add(index, element);
    }
    // when the index is inside the sorted section, the element will be added at the end
    else if (index >= 0 && index < nSorted) {
        super.add( index: size() - 1, element);
    }
}
```

Figure 1. A code snippet of the method add().

In the method *add()* we have used the index to calculate the position and the element that is to be placed in the list of elements. First we check if the given element has to be placed on a position that is outside of range of the list of the sorted elements. If this is the case, we add the element to the list and if there is an element on that position it will scoop that element (and all elements with bigger indexes) by one position. This will be done in the unsorted list, which is unsorted anyway. The order of these elements could be completely random. So we could have done just *super.add(element).* There also wasn't any test that could verify whether the given index was actually utilized. We created this test to make sure this is being verified. This means that without our custom test (*TestUnsortedIndexOfAddedElement()*), the given index in the *add()* function has no use. Either the element is being placed at a (random) position of choice in the unsorted part of the list or, as the next check makes sure, the element gets added to the end of the list of elements. However this is part of the assignment, for this is the reason we created a test to verify this variable. The next check is passed if the given index is positioned on the sorted part of the list. If that is the case, it must not temper with the order of the sorted list, which caused us to position the given element to the end of the list of elements.

3. nSorted = 0;

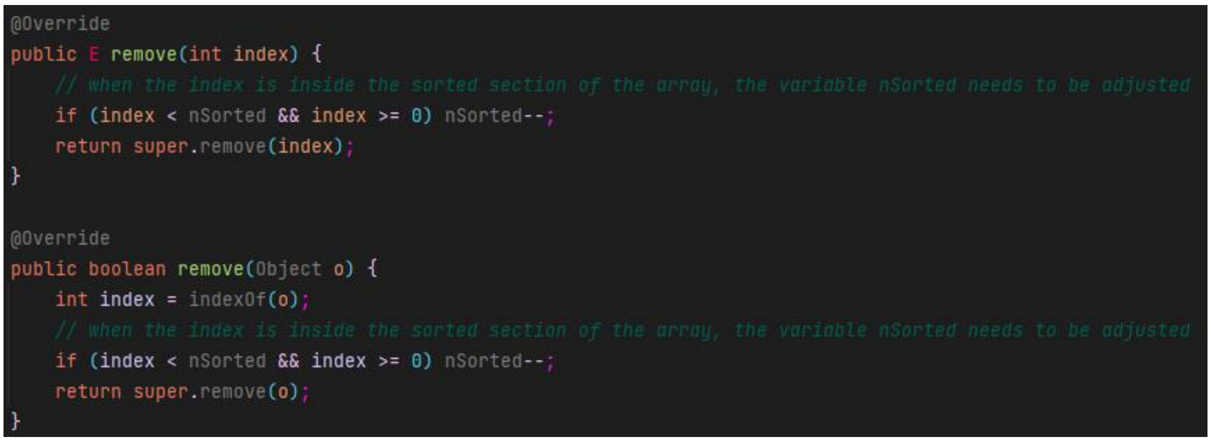


```java
@Override
public void sort(Comparator<? super E> c) {
    super.sort(c);
    this.ordening = c;
    this.nSorted = this.size();
}
```

Figure 2. A code snippet of the method sort().

When creating this documentation we saw that there is a couple of variables that are not really explained well. One of these examples is *nSorted.* For this documentation to be understandable, we think it is of importance that the term *nSorted* is explained as well. *nSorted* is being used quite a significant amount of times and therefore we wanted to explain its meaning.
We have created the protected integer nSorted in the beginning of the class OrderedArrayList as we needed to have a variable in which the amount of sorted elements within the used list would be stored. We can see in the snippet above that we are using two methods that come with the library of ArrayList, which are *sort()* and *size()*. With *sort()* we use ArrayLists' own method to sort the elements in the list correctly and save that sorted list in the variable *ordening.* Then because of the list being sorted, the amount of sorted elements within the list is known. The total list and the list of elements could actually be **bigger** than the amount of sorted elements in the list, so ArrayList has a great method called *size()* which returns the amount of elements within the used list, which we then store in the variable *nSorted* after the list has been sorted. In other words, the variable nSorted is used to store **the amount of sorted elements within the used list**.

4. remove(int index) & remove(Object o);

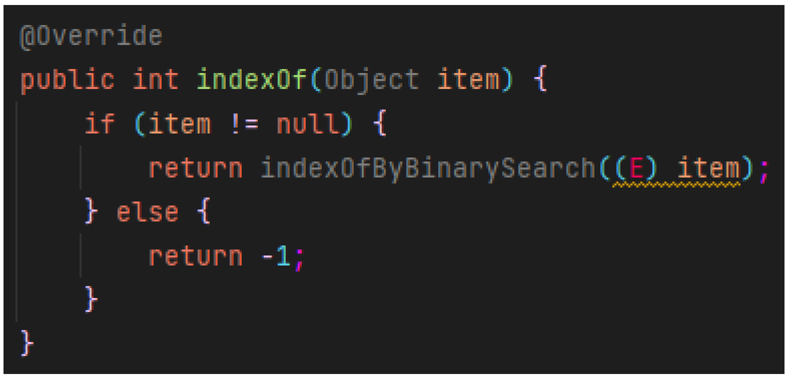


```java
@Override
public E remove(int index) {
    // when the index is inside the sorted section of the array, the variable nSorted needs to be adjusted
    if (index < nSorted && index >= 0) nSorted--;
    return super.remove(index);
}

@Override
public boolean remove(Object o) {
    int index = indexOf(o);
    // when the index is inside the sorted section of the array, the variable nSorted needs to be adjusted
    if (index < nSorted && index >= 0) nSorted--;
    return super.remove(o);
}
```

Figure 3. A code snippet of the methods remove() and remove().

We have created two methods called *remove*() which was part of the assignment. We needed to create two ways to remove an item from the list. One can be called by passing an integer (*index*) which basically tells the program to remove the item on that specific spot in the list. To sustain the representation invariant we made sure we reduce number which is represented by nSorted if the index is within the bounds of the sorted list. We have chosen to do this because of our experience through trial and error as we first got a couple of errors trying to run the test. When the limits are met, we just simply make the list of sorted elements smaller and we call the *remove()* of the ArrayList to remove the actual element. And before it is actually erased, the method in the ArrayList 'library' returns the old value which makes us able to trace what has been removed and to restore any faults that our written code may do. *remove(Object o)* basically does exactly the same with one exception. Because the user of the program wants a specific element deleted, we first had to find the place the element was stored in the list, so we first call the *indexOf()* method to find the *index* of the given element. The rest of the code within the method *remove(Object o)* does the same as the method *remove(int index)* as we found the index needed of the element that is asked to be removed.

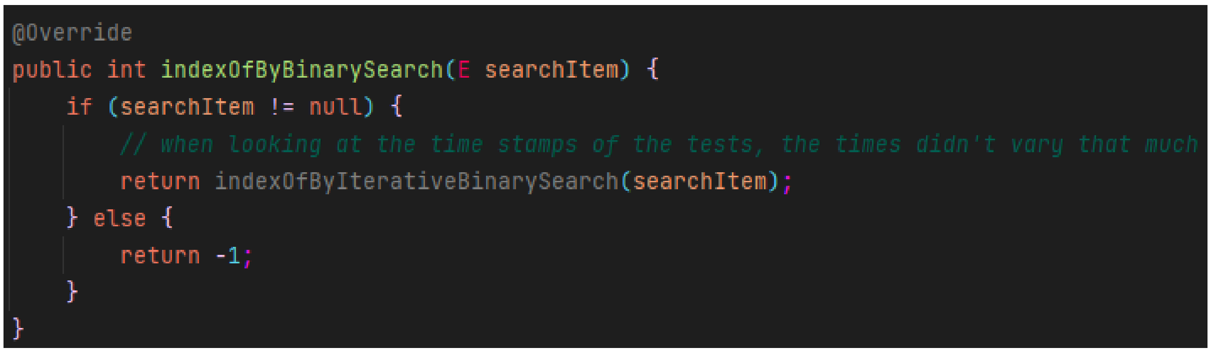
5. indexOf(Object item) & indexOfByBinarySearch(E searchItem);



```
@Override
public int indexOf(Object item) {
    if (item != null) {
        return indexOfByBinarySearch((E) item);
    } else {
        return -1;
    }
}
```

Figure 4. A code snippet of the method indexOf().

Now we are slowly getting to the good part of the code. The searches. In the methods *remove() and remove()* we have taken a small look into how to find a certain place within a list. We saw that when having an index, the search was being taken care of by the *remove()* method of ArrayList itself. As that is a method in ArrayList's own library we will not go into these details. However we needed to be able to search through the list ourselves as well. In *indexOf()* we are giving the element to find out which place within the list is dedicated to that element so that we can use the index of the element in different parts of the code. As we can see first a check is placed to see if there is an actual element given so that we don't try to find the index of a non-existent element. If no element has been given there is no index to give back, so as seen above we simply return with *-1*. However there was no test to verify this return value. So we created one, the *TestObjectsExistence()*. Otherwise we could have returned any random integer and the tests would run correctly.



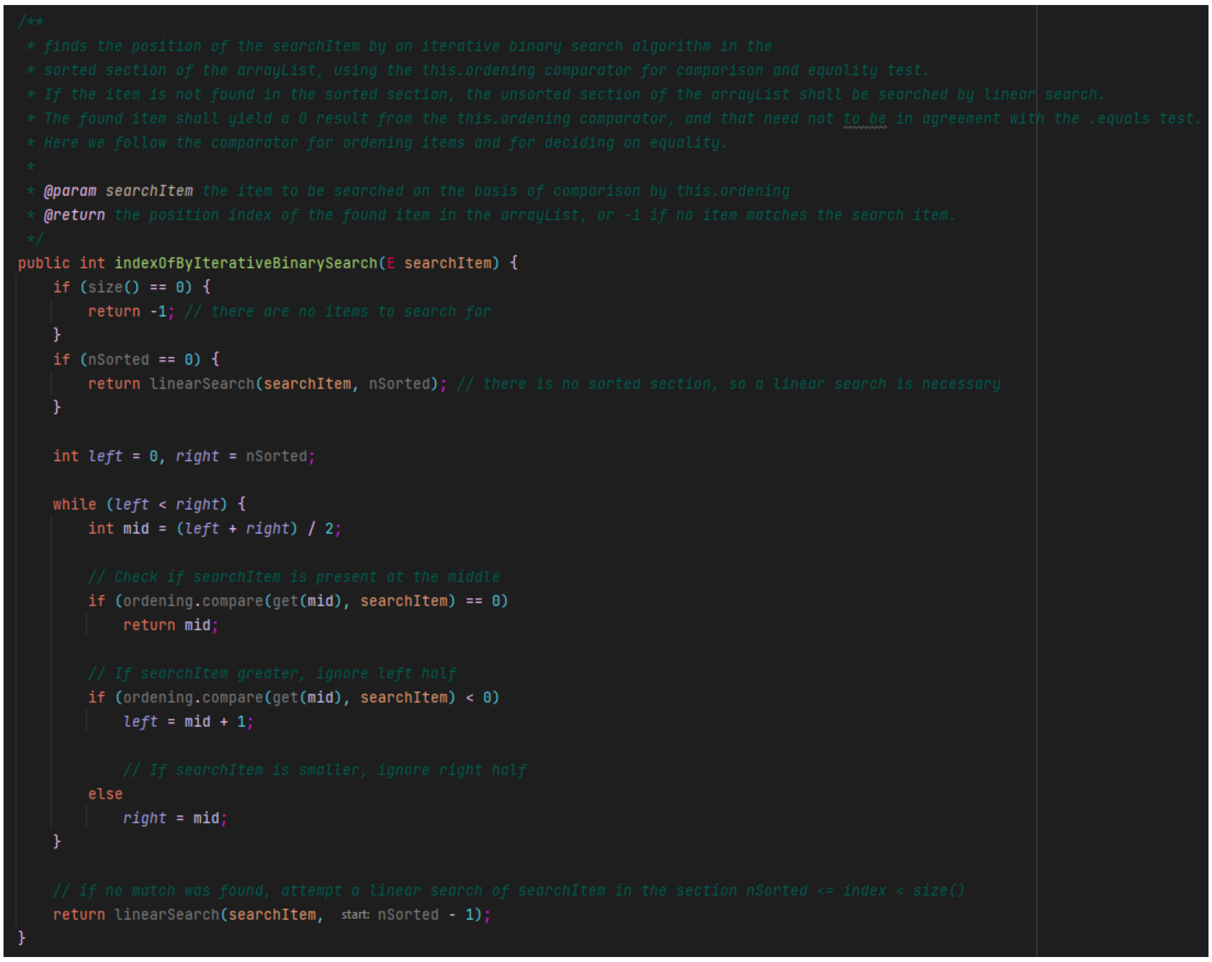
```
@Override
public int indexOfByBinarySearch(E searchItem) {
    if (searchItem != null) {
        // when looking at the time stamps of the tests, the times didn't vary that much
        return indexOfByIterativeBinarySearch(searchItem);
    } else {
        return -1;
    }
}
```

Figure 5. A code snippet of indexOfByBinarySearch().

If an element has been given (and is therefore not null) we call to the *indexOfByIterativeBinarySearch().* This could also call for the *indexOfByRecursiveSearch()* but as seen in the comment in the code snippet, the times did not vary that much.
This method was instantiated in the interface which made us require the implementation. We have discussed the actual code with some of our brightest classmates and found out that this code looks a lot like *indexOf()*. We are unsure as to why that is. As of now this is duplicate code as the *indexOf()* does exactly the same. The check is already done in *indexOf()* and the return can be placed in *indexOf()* instead of in the current method. We made our own tests and tried to utilize the

indexOfByBinarySearch() in these tests, but also these work when just calling for the *indexOf()* method as that method just calls for the *indexOfByBinarySearch().* As we think this is duplicate code, we thought it is unnecessary to put it in the interface and to make the method private. Which then takes away the checks, and the code will solely return *indexOfByIterativeBinarySearch()* (Or any other search if preferred). This will make the code really unnecessary as it will only contain a return, and *indexOf()* might as well just return *indexOfByIterativeBinarySearch()* on its own. However this was the assignment and we couldn't take it away.

6. indexOfByIterativeBinarySearch();

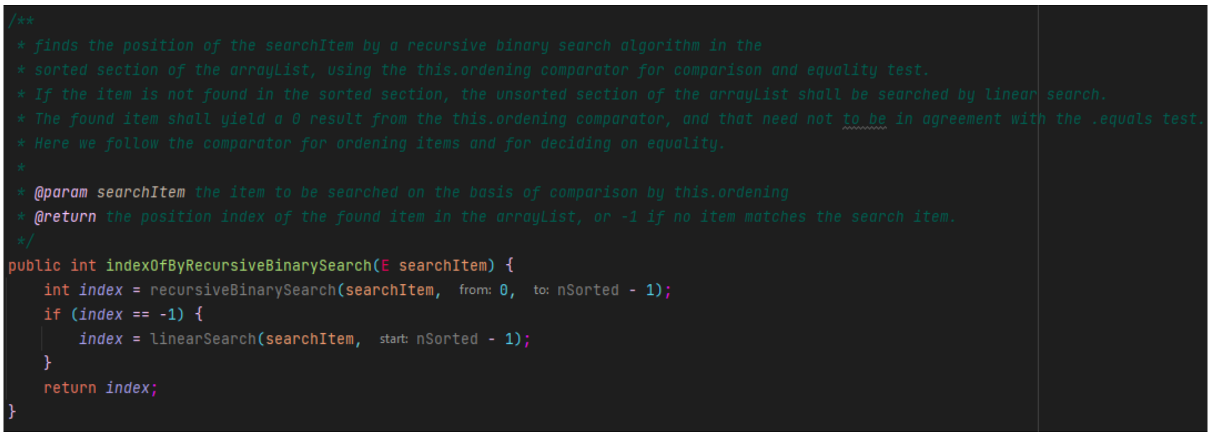


```java
/**
 * finds the position of the searchItem by an iterative binary search algorithm in the
 * sorted section of the arrayList, using the this.ordening comparator for comparison and equality test.
 * If the item is not found in the sorted section, the unsorted section of the arrayList shall be searched by linear search.
 * The found item shall yield a 0 result from the this.ordening comparator, and that need not to be in agreement with the .equals test.
 * Here we follow the comparator for ordening items and for deciding on equality.
 *
 * @param searchItem the item to be searched on the basis of comparison by this.ordening
 * @return the position index of the found item in the arrayList, or -1 if no item matches the search item.
 */
public int indexOfByIterativeBinarySearch(E searchItem) {
    if (size() == 0) {
        return -1; // there are no items to search for
    }
    if (nSorted == 0) {
        return linearSearch(searchItem, nSorted); // there is no sorted section, so a linear search is necessary
    }

    int left = 0, right = nSorted;

    while (left < right) {
        int mid = (left + right) / 2;

        // Check if searchItem is present at the middle
        if (ordening.compare(get(mid), searchItem) == 0)
            return mid;

        // If searchItem greater, ignore left half
        if (ordening.compare(get(mid), searchItem) < 0)
            left = mid + 1;

            // If searchItem is smaller, ignore right half
        else
            right = mid;
    }

    // if no match was found, attempt a linear search of searchItem in the section nSorted <= index < size()
    return linearSearch(searchItem, start: nSorted - 1);
}
```

Figure 6. A code snippet of the method indexOfByIterativeBinarySearch().

Above we see the method *indexOfByIterativeBinarySearch()* which is called in *indexOfByIterativeSearch().* We first had to create a check that checks if there any elements in the list, otherwise we would be searching through an empty list, without elements. Then it was necessary to make a check that verifies whether there is a sorted list, as binary search can only be applied to lists with a non-random order. If there is not a sorted list, we call upon the *linearSearch()* which we will discuss later. If there is a sorted list, we create a left side and a right side so we can find the middle of the list and check if the element exists in the lower half or the upper half. We do this iteratively as long as we do not find the element. When we do not find the element we redefine the left or the right side depending on the position of the element we are searching for. The reason we are doing this, is to make the search area smaller and smaller until we found the element by exclusion of all elements that we are not looking for. If this fails, we created a failsafe by calling the *linearSearch()* if the method above does not work so that it has multiple ways to find the index of the element we are searching for if one fails to get a result.

7. indexOfByRecursiveBinarySearch();

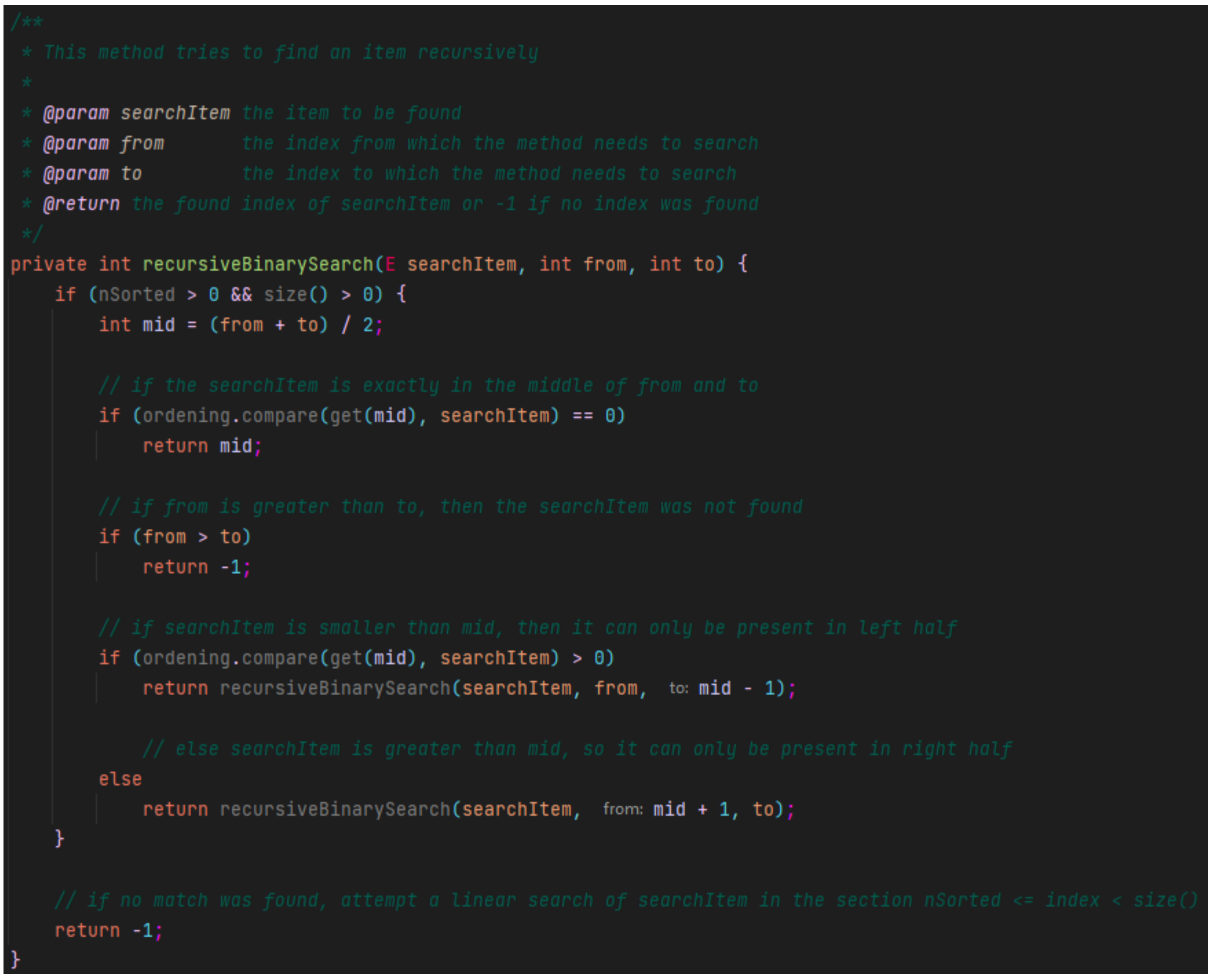


```java
/**
 * finds the position of the searchItem by a recursive binary search algorithm in the
 * sorted section of the arrayList, using the this.ordening comparator for comparison and equality test.
 * If the item is not found in the sorted section, the unsorted section of the arrayList shall be searched by linear search.
 * The found item shall yield a 0 result from the this.ordening comparator, and that need not to be in agreement with the .equals test.
 * Here we follow the comparator for ordening items and for deciding on equality.
 *
 * @param searchItem the item to be searched on the basis of comparison by this.ordening
 * @return the position index of the found item in the arrayList, or -1 if no item matches the search item.
 */
public int indexOfByRecursiveBinarySearch(E searchItem) {
    int index = recursiveBinarySearch(searchItem,  from: 0,  to: nSorted - 1);
    if (index == -1) {
        index = linearSearch(searchItem,  start: nSorted - 1);
    }
    return index;
}
```

Figure 7. A code snippet of the indexOfByRecursiveBinarySearch() method.

This method creates a integer *index* which is ultimately returned and will give the asked index of the right element. It first calls for the *recursiveBinarySearch()* which we will discuss later. We had to contain a check of the outcome of the *recursiveBinarySearch()* because that method returns *-1* if the element was not found. If that is the case, just like the *indexOfByIterativeBinarySearch()*, we created a failsafe by calling to the *linearSearch()*.

8. recursiveBinarySearch();

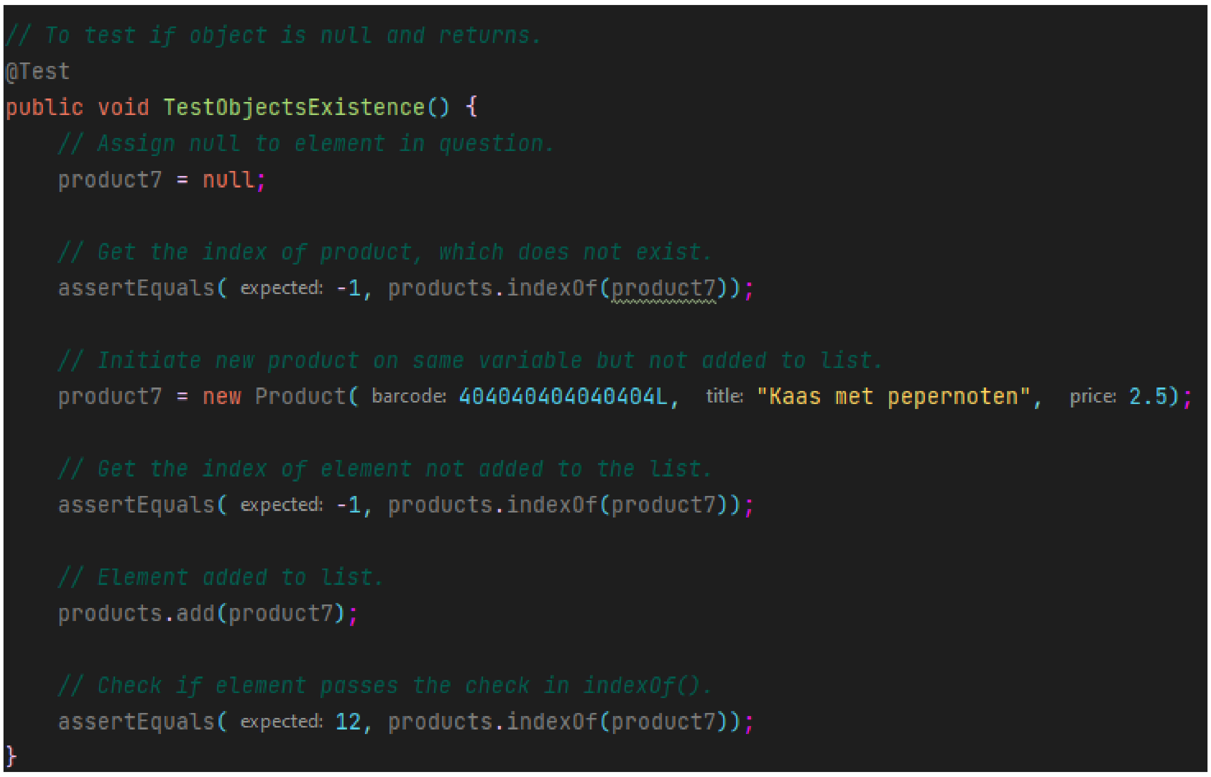


```java
/**
 * This method tries to find an item recursively
 *
 * @param searchItem the item to be found
 * @param from       the index from which the method needs to search
 * @param to         the index to which the method needs to search
 * @return the found index of searchItem or -1 if no index was found
 */
private int recursiveBinarySearch(E searchItem, int from, int to) {
    if (nSorted > 0 && size() > 0) {
        int mid = (from + to) / 2;

        // if the searchItem is exactly in the middle of from and to
        if (ordening.compare(get(mid), searchItem) == 0)
            return mid;

        // if from is greater than to, then the searchItem was not found
        if (from > to)
            return -1;

        // if searchItem is smaller than mid, then it can only be present in left half
        if (ordening.compare(get(mid), searchItem) > 0)
            return recursiveBinarySearch(searchItem, from,  to: mid - 1);

            // else searchItem is greater than mid, so it can only be present in right half
        else
            return recursiveBinarySearch(searchItem,  from: mid + 1, to);
    }

    // if no match was found, attempt a linear search of searchItem in the section nSorted <= index < size()
    return -1;
}
```

Figure 8. A code snippet of the method recursiveBinarySearch().

The *recursiveBinarySearch()* method does essentially the same as the *indexOfByIterativeBinarySearch()* apart from a few key differences. It starts the same but when we arrive at the *if-statements* that check if the item is positioned at the lower- or upper half, we return with the *recursiveBinarySearch()* method and replace the *to*-variable with the mid if the searched element is in the lower half and we replace the *from*-variable with the mid if the searched element is in the upper half. This way we make the method loop recursively until it has found the index of the element and returns the index value of that element. If the element is not found, the code returns *-1*. The result of this outcome has been discussed in chapter 8.

9. TestObjectsExistence();



```java
// To test if object is null and returns.
@Test
public void TestObjectsExistence() {
    // Assign null to element in question.
    product7 = null;

    // Get the index of product, which does not exist.
    assertEquals( expected: -1, products.indexOf(product7));

    // Initiate new product on same variable but not added to list.
    product7 = new Product( barcode: 404040404040404L,  title: "Kaas met pepernoten",  price: 2.5);

    // Get the index of element not added to the list.
    assertEquals( expected: -1, products.indexOf(product7));

    // Element added to list.
    products.add(product7);

    // Check if element passes the check in indexOf().
    assertEquals( expected: 12, products.indexOf(product7));
}
```

Figure 9. A code snippet of the test method TestObjectsExistence().

As discussed in chapter 6, we created a test to see if the object actually exists. As we were programming and testing we found out that the indexes of the objects are not actually tested. In the code snippet of chapter 6 we have seen a check to see if the element given is not null, but there was no test in the project that would actually verify if an element could be null. This is the reason we built this test, to check if the code returns the right value if the object does not exists or is not placed in the list. We do this by creating an element and not adding it to the list, yet checking if we can take it out of the list. It should return *-1.* Then if we add it to the list, we check if the product is added (to the right place) in the list. We did this to make sure our code would make sure the elements in the list exist and would run correctly.