1 Category Partition Method

1.1 Prerequisites Consideration

1.1.1 Type of data structure

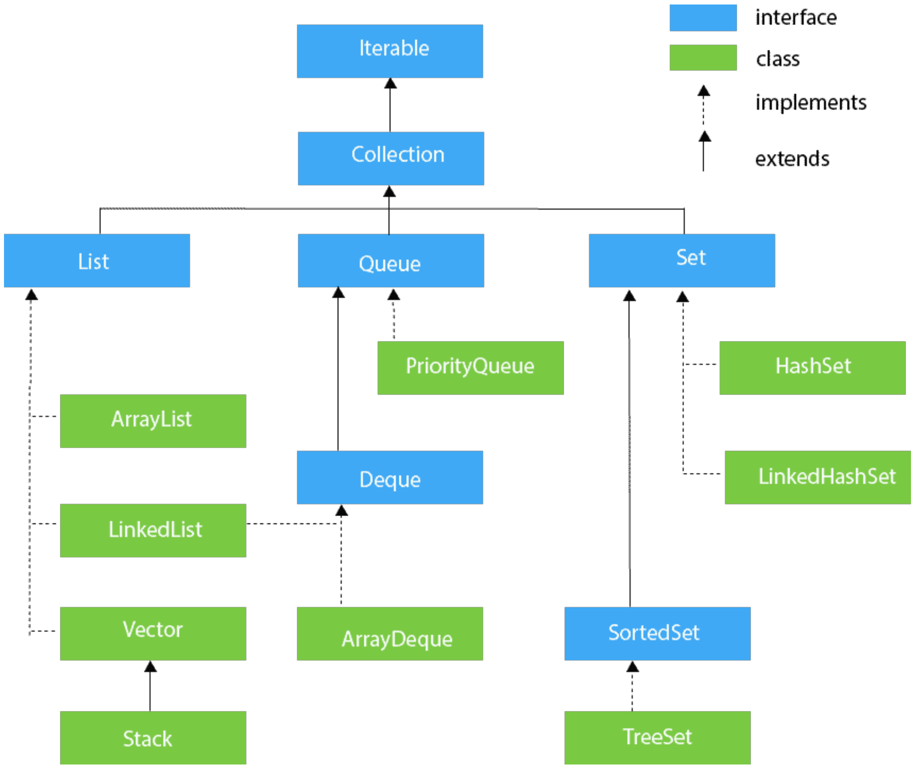

Figure 1: Java Collections Framework hierarchy [7].

As shown in **Figure 1**, `Collection` is an interface that is implemented by three different sub-interfaces, and each sub-interface is extended by different data structure classes. Therefore, it is worth to investigate the relationship between the type of data structure used and the output of the testing implementation.

1. `List`:

   (a) `ArrayList`, `LinkedList`, and `Vector` implement the `List` interface. The main difference between them is their implementation which causes different performance for different operations [1, 8, 12].

   (b) Both `Collections.sort` and `Collections.rotate` use interface over concrete types for the `list` parameter. This implies that both methods will only call methods that are defined by the `List` interface (so no `ArrayList` etc. specific methods).

   Hence, assume that `ArrayList`, `LinkedList`, and `Vector` have been correctly implemented, the testing results of `Collections.sort` and `Collections.rotate` should not be affected by the type of data structure used.

2. `Collection`:

   (a) `List`, `Queue`, and `Set` are fundamentally different from each other in terms of storing and manipulating the data. One crucial difference is that `Set` does not allow duplicate elements [11], which means that there will be certain categories that are not applicable when the input `coll` is a `Set`.

(b) `Collections.min` uses interface over concrete types for the `coll` parameter, which implies that the `min` method will only use methods that are defined by the `Collection` interface.

Assume that `List`, `Queue`, and `Set` have been correctly implemented, the testing results of `min` should not be affected by the type of data structure used as well.

1.1.2 Type of elements

Another aspect to consider is the type of the elements stored in the `List` or `Collection`. Since, all elements must be an instance of a class that implements the `Comparable` interface [4], the correctness of `Collections.sort` and `Collections.min` is then dependent on the implementation of the `compareTo` method.

In this testing assessment, the primary objective is to test the `sort`, `rotate`, and `min` methods of the `Collections` interface. The premise for this assessment is that all classes that implemented the `Comparable` interface have defined their respective `compareTo` method correctly, otherwise it would have contradict with the expected output results set for the test cases. Hence, it is assumed that the type of elements used in the testing (e.g. `Integer`, `Double`, `Date` etc.) would not affect the test results, and it would not be taken into consideration as one of the category partition for test inputs.

1.2 Parameter: List

N.B. All elements in the list must be an instance of a class that implements the `Comparable` interface and must be mutually comparable.

Category L1 Empty list

Input: `list = []`

Reason: One of the edge cases of the `List` interface. Given this as the input, both `Collections.sort` and `Collections.rotate` should return an empty list as the output.

Category L2 Single element in the list

Input: `list =` $[x]$

Reason: One of the edge cases of the `List` interface. The output returned by both `Collections.sort` and `Collections.rotate` should be the same as the input.

Category L3 More than one element in the list

Input: `list =` $[x_0, x_1, \ldots, x_n]$ for $n > 1$, where n is the total number of elements in the list

Reason: The is the minimum viable case for testing the actual functionality of both `Collections.sort` and `Collections.rotate`. This is the base form that is used to define other variants of the input (**Category L4 - L7**).

Category L4 Repeated elements in the list

Input: `list = `$[x_0, x_1, \ldots, x_n]$ for $n > 1$. In addition, there exists some a, b for all $0 \leq a, b \leq n$, the value of x_a is equal to x_b.

Reason:

1. This is to test the stability of the `Collections.sort` sorting algorithm. The sorting algorithm should sort the repeated elements in the same order that they appear in the input.
2. This is to test that `Collections.rotate` only modifies the position of the elements. The value of each element should not affect the final output.

Category L5 List contains elements of different class

Input: `list = `$[x_0, x_1, \ldots, x_n]$ for $n > 1$, there exists some i, j such that for all $0 \leq i, j \leq n,\ i \neq j$, the type of $x_i \neq$ type of x_j.

Reason: This is a special category for `Collections.rotate`. Since the purpose of `Collections.rotate` is to modify the index of each element in the list, the actual value of each element should have no significant effect for the method. Therefore, the `rotate` method should work correspondingly when the list contains elements of different types.

Category L6 List is sorted in ascending natural order of its elements

Input: `list = `$[x_0, x_1, \ldots, x_n]$ for $n > 1$, where the list is arranged according to the natural ordering [4] of its elements such that $x_k \leq x_{k+1}$ for all $0 \leq k \leq n$. The list can contain repeated elements.

Reason: This is to test the sorting stability of `Collections.sort`. Since the input list is already sorted, then the elements in the output list should have the same order as the input list.

Category L7 List is sorted in descending natural order of its elements

Input: `list = `$[x_0, x_1, \ldots, x_n]$ for $n > 1$, where the list is arranged according to the **reverse** natural ordering [4] of its elements such that $x_k \geq x_{k+1}$ for all $0 \leq k < n$.

Reason: This is the inverse case of **Category L6**, where the input list is sorted in reverse order. The aim is to test the sorting stability of `Collections.sort` and also ensure that it can handle the cases where certain parts of the list (i.e. sub-list) are inversely sorted.

Category L8 List size is large

Input: `list = `$[x_0, x_1, \ldots, x_n]$ for $n > 200$

Reason: According to the documentation of `Collections.rotate` [3], the method uses two different algorithms depending on the list size (another condition is related to `RandomAccess` interface, which is not considered). The aim is to validate that both algorithms are able to correctly rotate the input list by the specified distance.

1.3 Parameter: distance

Category D1 Negative number

Input: `distance < 0`

Reason: To ensure that the `Collections.rotate` method will work with negative distance input by covering the negative domain of `int` data type.

Category D2 Zero

Input: `distance == 0`

Reason: 0 is the default value of `int` data type in Java, and also the starting index value of `List`. This is to ensure that the `Collections.rotate` method will work when the input distance is zero.

Category D3 Positive number

Input: `distance > 0`

Reason: To ensure that the `Collections.rotate` method will work with positive distance input by covering the positive domain of `int` data type.

Category D4 Larger than list size

Input: `distance > list.size()`

Reason: A derivative of **Category D3**. Since the `rotate` method uses modulo operation on the input distance [3], this is to test whether the behaviour of inputting a distance larger than `list.size()` is the same as inputting the value of `distance % list.size()` [3].

Category D5 Equal to list size

Input: `distance == list.size()`

Reason: This is to validate that if the distance is equal to the `list.size()`, then `Collections.rotate` will return the the same output as the input.

Category D6 Smaller than list size

Input: `distance < list.size()`

Reason: A derivative of **Category D1 - D3** and the inverse case of **Category D4**. This is to validate the assumption that there exists some d for all $d >$ `list.size()` and n for all $n <$ `list.size()` such that `rotate(list, d) == rotate(list, n)`.

Category D7 Equal to minimum boundary value of `int`

Input: `distance == Integer.MIN_VALUE`

Reason: The minimum value an `int` can have in Java is -2^{31} [6]. The aim is to validate the assumption that if `Collections.rotate` works for the minimum value of `int`, then it should work correctly for any value **larger than the minimum value**.

Category D8 Equal to maximum boundary value of `int`

Input: `distance == Integer.MAX_VALUE`

Reason: The maximum value an `int` can have in Java is $2^{31} - 1$ [5]. The aim is to validate the assumption that if `Collections.rotate` works for the maximum value of `int`, then it should work correctly for any value **smaller than the maximum value**.

1.4 Parameter: Collection

N.B. All elements in the collection must be an instance of a class that implements the `Comparable` interface and must be mutually comparable.

Category C1 Single element in collection

Input: `coll` $= \{x\}$

Reason: Edge case of the `Collections.min` method. The method should only return the single element as the minimum element of the given collection.

Category C2 More than one element in collection

Input: `coll` $= \{x_0, x_1, \ldots, x_n\}$ for $n > 1$

Reason: To test whether the `Collections.min` method is able to find and return the minimum element of the given collection.

Category C3 Repeated elements in collection

Input: `coll` $= \{x_0, x_1, \ldots, x_n\}$ for $n > 1$. In addition, there exists some a, b for all $0 \leq a, b \leq n,\ a \neq b$, the value of x_a is equal to x_b.

Reason: To test whether the `Collections.min` method is able to handle repeated elements and correctly return the minimum element of the given collection.

Category C4 Repeated minimum elements in collection

Input: `coll` $= \{x_0, x_1, \ldots, x_n\}$ for $n > 1$. In addition, there exists some a, b for all $0 \leq a, b \leq n,\ a \neq b$, the value of x_a is equal to x_b and both x_a and x_b are the minimum elements in the collection.

Reason: A derivative of **Category C3** to test the stability of `Collections.min` method. It is expected that the method will treat the repeated minimum elements as the same element and return correctly.

Category C5 Collection contains the minimum possible value of the class

Input: `coll` $= \{x_0, x_1, \ldots, x_n\}$ for $n > 1$, and there exists a k where $0 \leq k \leq n$ such that x_k is the minimum possible value of the element's class.

Reason: To cover the boundary case of `Collections.min`, and to test whether the method always return the minimum possible value of the class when the given collection contains it.

Category C6 Collection is in ascending natural order of its elements

Input: `coll` $= \{x_0, x_1, \ldots, x_n\}$ for $n > 1$, where the collection is arranged according to the natural ordering [4] of its elements such that $x_k \leq x_{k+1}$ for all $0 \leq k \leq n$. The collection can contain repeated elements.

Reason: Since the given collection is already sorted, then the minimum element returned should have the same value as first element of the collection.

Category C7 Collection is in descending natural order of its elements

Input: `coll` $= \{x_0, x_1, \ldots, x_n\}$ for $n > 1$, where the collection is arranged according to the natural ordering [4] of its elements such that $x_k \geq x_{k+1}$ for all $0 \leq k < n$. The collection can contain repeated elements.

Reason: The inverse case of **Category C6**. Since, the given collection is sorted in reverse order, then the minimum element returned should have the same value as the last element of the collection.

2 Test Cases

2.1 `Collections.sort(List<T> list)`

1. **Test Case 1**

   Categories: L2

   Input: `list = [1]`

2. **Test Case 2**

   Categories: L3 $\wedge$ L4

   Input: `list = [6, 8, 2, 4, 7, 5, 3, 2, 9]`

3. **Test Case 3**

   Categories: L3 $\wedge$ L7

   Input: `list = [9, 8, 7, 6, 5, 4, 3, 2, 1]`

2.2 `Collections.rotate(List<?> list, int distance)`

1. **Test Case 1**

   Categories: L1, D2 $\wedge$ D5

   Input: `list = [], distance = 0`

2. **Test Case 2**

   Categories: L3 $\wedge$ L5, D3 $\wedge$ D4 $\wedge$ D8

   Input: `list = [1, 2, ‘a’, ‘b’, 3.7, 4, "str"], distance =` $2^{31} - 1$

3. **Test Case 3**

   Categories: L3 $\wedge$ L8, D1 $\wedge$ D6 $\wedge$ D7

Input: `list = [1, 2, ..., 200, 201]`, `distance =` -2^{31}

2.3 `Collections.min(Collection<? extends T> coll)`

1. **Test Case 1**

   Categories: C1

   Input: `coll` = {`1`}

2. **Test Case 2**

   Categories: C2 $\wedge$ C7

   Input: `coll` = {`1, 2, 3, 4, 5, 6, 7, 8, 9`}

3. **Test Case 3**

   Categories: C2 $\wedge$ C3 $\wedge$ C4 $\wedge$ C5

   Input: `coll` = {`6,` -2^{31}`, 2, 4, 7,` -2^{31}`, 5, 3, 2, 9`}

3 Metamorphic Relations

3.1 `Collections.sort(List<T> list)`

1. Reverse the original input list

   Input transformation: `Collections.reverse(listTransformed)`

   Relation after transformation: `listTransformed == list`

2. Double the size and content of original input list by adding itself

   Input transformation: `listTransformed.reverse(listTransformed)`

   Relation after transformation:

   (a) `listTransformed.size() == 2 * list.size()`

   (b) `listTransformed[2n] == list[n]`

   (c) `listTransformed[2n + 1] == list[n]`

3.2 `Collections.rotate(List<?> list, int distance)`

1. Reverse the original input list, and convert the distance to opposite sign after modulo with the list size

   Input transformation:

   (a) `Collections.reverse(listTransformed)`

   (b) `distanceTransformed = -(distance % list.size())`

   Relation after transformation: `listTransformed` is the reverse of `list`. After reversing `listTransformed` again, `listTransformed == list`

2. Decrease distance to observe the index different of each element

   Input transformation: `distanceTransformed < distance % list.size()`

   Relation after transformation: Every element in `listTransformed` is Δd indices "behind" `list`, where Δd is the difference between `distanceTransformed` and `distance % list.size()`

N.B.

1. For both relations, if the original list is empty, then `distanceTransformed = 0`.
2. Example for second relation: `rotate([1,2,3,4,5], 3)` will yield `[3,4,5,1,2]`; After input transformation, `rotate([1,2,3,4,5], 1)` will yield `[5,1,2,3,4]`.

3.3 `Collections.min(Collection<? extends T> coll)`

1. Add an element smaller than the minimum element in the original collection

   Input transformation: `collTransformed.add(`y`)`, where y is a random generated element

   Relation after transformation: `outputTransformed < output`

2. Remove all minimum elements from the original collection

   Input transformation: `collTransformed.removeIf(output::equals)`

   Relation after transformation: `outputTransformed > output`

N.B.

1. For the first relation, if the minimum element in the original collection is already the minimum possible element for the class, then the new element added is equal to the minimum element in the original collection. The relation will then be `outputTransformed == output`.
2. For the second relation, if all elements in the original collection are the minimum elements (i.e. duplicate), then removing them will cause `collTransformed` to be an empty collection. In this case, the relation is `outputTransformed == output` as there are no elements greater than the minimum element in the original collection.

4 Remarks

4.1 Outcome of the Tests

All three methods have managed to pass all their respective test cases, and all transformed outputs are proved to obey their respective metamorphic relation. As a result, it is suggested that all three methods are correct and complete.

1. `Collections.sort`: The method is able to rearrange the original list into ascending order, according to the natural ordering of its elements for all test cases. For all transformed inputs, the method is able to satisfy the expected relations.
2. `Collections.rotate`: The two metamorphic relations have shown that this method alters the index but not the value of each element in the list. After the rotation,

all elements in the list are preserved, but each of them is moved by the specified distance.

3. `Collections.min`: This method is assumed to be the least possible one to generate errors as it only performs read operation to find the minimum element [2]. After applying the metamorphic relations to alter the minimum element in the list, the method is still able to find and return the next minimum element in the list.

4.2 Likelihood of Faults

After searching the CVE archive for Oracle JDK [10] and Oracle JRE [9], no vulnerabilities that are related to these three methods have been found. Hence, it is suggested that all three methods are well implemented by the developers and have a low likelihood of failure.

References

[1] *Arraylist (java platform se8)*, Oracle Corporation, 6th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.

[2] *Collections min (java platform se8)*, Oracle Corporation, 5th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#min-java.util.Collection-`.

[3] *Collections rotate (java platform se8)*, Oracle Corporation, 5th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#rotate-java.util.List-int-`.

[4] *Comparable (java platform se8)*, Oracle Corporation, 6th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`.

[5] *Integer max value*, Oracle Corporation, 5th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#MAX_VALUE`.

[6] *Integer min value*, Oracle Corporation, 5th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#MIN_VALUE`.

[7] *Java collection class and interface hierarchy*, JavaTpoint, 4th Dec. 2019. [Online]. Available: `https://www.javatpoint.com/collections-in-java`.

[8] *Linkedlist (java platform se8)*, Oracle Corporation, 6th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html`.

[9] *Oracle jdk cve search results*, MITRE, 10th Dec. 2019. [Online]. Available: `https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-19117/Oracle-JRE.html`.

[10] *Oracle jre cve search results*, MITRE, 10th Dec. 2019. [Online]. Available: `https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-19116/Oracle-JDK.html`.

[11] *Set (java platform se8)*, Oracle Corporation, 6th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/Set.html`.

[12] *Vector (java platform se8)*, Oracle Corporation, 6th Dec. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html`.