Metamorphic Testing Assignment Report

150149370

December 2019

Contents

Chapter 1

Category Partition Method

This section details the categories that were chosen, and why.

1.1 List

1.1.1 Odd Number of Elements [ID: 01]

In JDK 8, the sort method uses a highly optimised mergesort to order lists. Since mergesort is a 'divide and conquer' algorithm, the behaviour around dividing odd length lists is investigated.

1.1.2 Linked List Type [ID: 02]

Allows us to probe the behaviour towards doubly linked lists, particularly because you can not directly access individual list elements (i.e. is not randomAccess) and the rotation algorithm is different for lists that do not support random access

1.1.3 Recurring values present [ID: 03]

Using recurring values allows for the probing of the sort algorithm, especially in terms of how it compares values.

1.1.4 Empty list element present [ID: 04]

We attempt to discover how the functions in question handle empty fields, and their rankings.

1.1.5 String List Element type [ID: 05]

For testing how the algorithms responds to the string data type.

1.1.6 Long List Element type [ID: 06]

For testing how the algorithms responds to long data types, and decimal values.

1.1.7 Negative values present [ID: 07]

We explore whether negative values induce unwarranted types of behaviour in the sort method, such as the use of absolute values during comparisons.

1.1.8 Long List Length [ID: 08]

The rotate function uses a different algorithm for large lists that do not support random access. 'Large' here is unspecified, but we will arbitrarily consider large as including upwards of 256 elements.

1.1.9 ArrayList List Type [ID: 09]

ArrayList facilitates random access, thus providing insight into how the functions respond to such structures.

1.2 Distance

1.2.1 Negative Distance [ID: 10]

According to the documentation, a negative distance should result in a forwards rotation. We explore this behaviour.

1.2.2 Distance $<=$ List Length [ID: 11]

We anticipate some periodicity in the rotation. This attempts to see how the function copes with a rotation larger than 1 full cycle.

1.2.3 Distance $=$ List Length [ID: 12]

A rotation distance equal to list length should not result in any change.

1.2.4 Distance Multiple of List Length [ID: 13]

Any rotation that is a multiple of the list length should also not result in any change.

1.2.5 Distance = 0 [ID: 14]

A distance of 0 should result in no change.

1.2.6 Positive Distance [ID: 15]

To explore the full breadth of polarity in the function.

1.3 Collection

1.3.1 Hash Set Collection Type [ID: 16]

Hash sets implement hashing and therefore do not guarantee ordering. It seems logical to explore how the minimum function handles hashed sets.

1.3.2 Tree Set Collection Type [ID: 17]

Tree sets are sorted sets and are thus form a different platform to test the Min function with.

1.3.3 Hash Map Collection Type [ID: 18]

Maps require a key and value, allowing us to test the function from yet another angle. Hash maps also have an unordered structure.

1.3.4 Multiple Minima [ID: 19]

Exploring the collections response to multiple minima could highlight comparison issues.

1.3.5 No Minimum [ID: 20]

Any collection with equal elements is effectively a 'straight line' having no minimum. Insight can be gained into how the minimum is selected when all are the same.

1.3.6 Negative Values [ID: 21]

We expect negative values to be considered low than their positive counterparts, even if they have larger absolute value.

1.3.7 Linked List Collection Type [ID: 21]

Chapter 2

Test Cases

This section details the test cases to be used.

2.1 Collections.sort

2.1.1 Test Case 1.1

Description: Linked list with an odd number of longs, some of which are repeated values, and some of which are negative.

Categories Used: 2,1,6,3,7

Concrete Input: The input is created as shown in Figure 2.1. Input arguments are 50 and 5 respectively.

```java
// CATEGORIES 2,1,6,3,7
public static LinkedList<Double> lListGen1(int listLength, int maxRepeats) {

    Random r = new Random();

    LinkedList<Double> listofDoubles = new LinkedList<Double>(); // Creates list that will eventually be returned

    while (listofDoubles.size() <= listLength) { // Loop stops when listLength is reached
        Double ranDouble = r.nextDouble(); // A random double is created
        int decider = r.nextInt(10); // A random integer who's value will determine whether to add a repeat or negative is created.
        int repeatTimes = r.nextInt(maxRepeats); // A random amount of repeat times, no more than the max specified is created.

        if (r.nextInt(7) > 4) { // Introduce negative values randomly
            ranDouble = ranDouble*-1;
        }

        if (decider > 5) { // If the repeat is triggered, the same value is added to the list severally.
            for (int numberOfRepeats = 0; numberOfRepeats <= repeatTimes; numberOfRepeats ++) {
                int addAtIndex = r.nextInt(listofDoubles.size()+1); // An int that will determine where to add a repeat is created
                listofDoubles.add(addAtIndex,ranDouble);
            }
        }
        else {
            listofDoubles.add(ranDouble);   // If no repeat is triggered, a random double is added to the end of the list.
        }
    }
    return listofDoubles;

}
```

Figure 2.1: Linked list generator 1 method

2.1.2 Test Case 1.2

Description: Array list of 301 strings.

Categories Used: 9,1,5

Concrete Input: The string generation method can be seen in figure 2.2. The two input arguments are 301, and 20 respectively.

```java
public class stringGenerator {
    // CATEGORIES 9,1,5
    public static ArrayList<String> generate(int listLength, int elementMaxLength) {
        Random r = new Random(); // Create a random object

        ArrayList<String> listofStrings = new ArrayList<String>(); // This is the list that will eventually be returned

        String AlphaNumericString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                + "0123456789"
                + "abcdefghijklmnopqrstuvxyz"; // We take values from this string.

        while (listLength-- != 0) { // Cycle through each potential element in the final list

            StringBuilder builder = new StringBuilder(elementMaxLength); // Create a blank builder that we will add characters to

            // Determine a random element length, less than the max and cycle through that number
            for (int index = 0; index <= r.nextInt(elementMaxLength); index ++) {

                // Add a random character from the string bank to the blank string
                builder.append(AlphaNumericString.charAt(r.nextInt(AlphaNumericString.length())));
            }

            listofStrings.add(builder.toString()); // Add the newly created string to the array
        }

        System.out.println(listofStrings);



        return listofStrings;
    }
```

Figure 2.2: String generator method

2.1.3 Test Case 1.3

Description: ArrayList of 400 strings with multiple empty list elements.

Categories Used: 4,5,8,9

Concrete Input: Creation method is shown in figure 2.3.

2.2 Collections.rotate

These test cases will borrow the corresponding lists created for the Sort test cases. For example, the list produced in test case 1.1 is reused in test case 2.2 with the specified rotation distance.

2.2.1 Test Case 2.1

Description: Negative rotation distance.

Categories Used: 10

```java
// CATEGORIES 4,5,8,9
public static ArrayList<String> generate2(int listLength, int elementMaxLength) {
    Random r = new Random(); // Create a random object

    ArrayList<String> listofStrings = new ArrayList<String>(); // This is the list that will eventually be returned

    String AlphaNumericString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            + "0123456789"
            + "abcdefghijklmnopqrstuvxyz"; // We take values from this string.

    while (listLength-- != 0) { // Cycle through each potential element in the final list

        StringBuilder builder = new StringBuilder(elementMaxLength); // Create a blank builder that we will add characters to

        if(r.nextInt(10) > 5) {
            // Determine a random element length, less than the max and cycle through that number
            for (int index = 0; index <= r.nextInt(elementMaxLength); index ++) {

                // Add a random character from the string bank to the blank string
                builder.append(AlphaNumericString.charAt(r.nextInt(AlphaNumericString.length())));
                listofStrings.add(builder.toString()); // Add the newly created string to the array
            }
        }
        else {
            listofStrings.add("");
        }
    }

    return listofStrings;
}
```

Figure 2.3: Second string generator method

Concrete Input: Randomly generated negative integer in a range of 0-100 (input = 100). Method is shown in figure 2.4.

```java
// CATEGORIES: 10
public static int negIntGen(int range) {
    Random r = new Random();
    int randomNum = r.nextInt(range)*-1;
    return randomNum;
}
```

Figure 2.4: Negative integer generator

2.2.2 Test Case 2.2

Description: Rotation distance larger than the list length.

Categories Used: 12

Concrete Input: Randomly generated number, larger than list size. Can be up to 5 times the list size. Method shown in figure 2.5.

2.2.3 Test Case 2.3

Description: Rotation is a multiple of the list length, positive or negative.

Categories Used: 12,15

Concrete Input: Uncapped multiple of the list size. Method shown in figure 2.6.

```java
    // CATEGORIES: 12, 15
    public static int IntGen(int minSize) {
        Random r = new Random();
        int randomNum = r.nextInt(minSize*r.nextInt(4))+minSize;
        return randomNum;
    }
```

Figure 2.5: Positive integer generator

```java
    // CATEGORIES: 12, 15
    public static int multipGen(int listLength) {
        Random r = new Random();
        int randomNum = listLength*r.nextInt();
        return randomNum;
    }
```

Figure 2.6: Multiple integer generator

2.3 Collections.min

2.3.1 Test Case 3.1

Description: Hash map of different keys, all with the same integer value.

Categories Used: 18,20

Concrete Input: Method for the creation of this map is shown in figure 2.7.

```java
    // CATEGORIES 18,20
    public static HashMap hmapGen(int mapSize, int consVal){
        HashMap<String, Integer> hashy = new HashMap<String, Integer>();

        Random r = new Random();
        String AlphaNumericString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                + "0123456789"
                + "abcdefghijklmnopqrstuvxyz"; // We take values from this string.
        while (mapSize-- != 0) {
            int keyLength = r.nextInt(5)+1; // Set the max key length to 6 and min to 1.
            StringBuilder builder = new StringBuilder(keyLength); // Create a blank builder that we will add characters to
            for (int index = 0; index <= keyLength; index ++) {
                // Add a random character from the string bank to the blank string
                builder.append(AlphaNumericString.charAt(r.nextInt(AlphaNumericString.length())));
            }

            hashy.put(builder.toString(), consVal); // make an entry to the map with constant value and the new string as the key.

        }
        return hashy;

    }
```

Figure 2.7: Hash map generator

2.3.2 Test Case 3.2

Description: Tree set with negative values.

Categories Used: 17,21

Concrete Input: Method for the creation of this map is shown in figure 2.8.

```java
    // CATEGORIES 17,21
    public static TreeSet<Integer> tset(int setLength){
        Random r = new Random();
        TreeSet<Integer> treeSet = new TreeSet<Integer>();

        while(setLength-- !=0) {
            treeSet.add(r.nextInt());
        }
        return treeSet;

    }
```

Figure 2.8: Tree set generator

2.3.3 Test Case 3.3

Description: Linked List with multiple minima.

Categories Used: 22,19

Concrete Input: Method for the creation of this map is shown in figure 2.9.

```java
    // CATEGORIES 22,19
    public static LinkedList treeSetGen(int setSize, int minVal){
        LinkedList<Integer> hashy = new LinkedList<Integer>();

        Random r = new Random();

        while (hashy.size() <= setSize) {
            int ranNum = r.nextInt();
            if (ranNum < minVal) {
                ranNum = ranNum + Math.abs(ranNum) + Math.abs(r.nextInt());
            }
            int decider = r.nextInt(20);

            if(decider>10) {
                ranNum = minVal;
            }
            hashy.add(ranNum);
        }
        return hashy;


    }
```

Figure 2.9: Linked List generator for multiple minima

Chapter 3

Metamorphic Relations

3.1 Collections.sort

Input Transform	Output Relationship
$x' = rev(x)$	$z == z'$
$x' = substr(x)$	$z' \subseteq z$

3.2 Collections.rotate

Input Transform	Output Relationship
$x' = x + listLength$	$z == z'$
$x' = x - x$	$z == z'$

3.3 Collections.min

Input Transform	Output Relationship
$x' = 2x$	$z ==' 2z$
$x' = x + a$	$z' = z + a$

Chapter 4

Outcomes

dfrefw