Guidebook: Writing Java Code from Test Code

Table of Contents

1. Introduction

This guidebook will help you learn how to write source code based on test code. You will practice analyzing test cases, writing the required implementation (source code), and verifying your solution using Visual Studio Code.

2. Simple Example

This section gives you a complete walk-through of how to read and understand a JUnit test, and how to write the matching source code.

We use a simple calculator class to demonstrate this process clearly.

When we build a program (like a sorting algorithm, searching algorithms, etc), we should make sure:

- It works correctly
- It doesn't break when something changes
- It handles errors properly

2.1 Problem Sample

That's where **test codes** come in—they help you automatically check whether our **main program behaves as expected**.

Test code for Calculator class
CalculatorTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import org.junit.Test;
 5
 6  public class CalculatorTest {
 7
 8      @Test
 9      public void testAdd() {
10          Calculator calc = new Calculator();
11          int codeInput1 = 2;
12          int codeInput2 = 3;
13          int codeOutput = calc.add(codeInput1, codeInput2);
14          int expectedOutput = 5;
15
16          try {
17              assertEquals("Addition test failed:", expectedOutput, codeOutput);
18          } catch (AssertionError ae) {
19              System.out.println(ae);
20          }
21      }
22  }
```

From the test case we create a class with the exact class and method names so that the test passes successfully.

The following is an explanation of each part of the test case:

1. `package dataStructure;`

   The statement in line 1 places the file in a package named `dataStructure`.
   Packages are used to organize Java classes into namespaces or modules.
   Test cases and source code must be in the same package as illustrated below.

   ```
   project-folder/
   └─── src/
     └─── dataStructure/
       ├─── Calculator.java
       ├─── CalculatorTest.java
   ```

2. `import static org.junit.Assert.*;`

   The import in line 3 imports all static methods from the Assert class in the JUnit framework — including:
   - assertEquals()
   - assertTrue(),
   - assertFalse()

3. `import org.junit.Test;`

   The import in line 4 needed when we use @Test annotation.

4. `public class CalculatorTest {`

   Statement in line 6 defines a test class named CalculatorTest.
   It will contain unit tests to verify the functionality of the Calculator class.

5. `@Test`

   Annotation in line 8 tells JUnit that the method below is a test method, which should be executed when tests run.

6. `public void testAdd() {`

   Statement in line 9 declares the test method called `testAdd`.

7. `Calculator calc = new Calculator();`

Statement in line 10 creates an object from the `Calculator` class so that you can call its methods.

8. `int codeInput1 = 2;`
9. `int codeInput2 = 3;`

Statements in line 11 and 12 are the input values for the `add` method.
Using separate variables for inputs is helpful for clarity and debugging.

10. `int codeOutput = calc.add(codeInput1, codeInput2);`

Statement in line 13 calls the `add` method with the given inputs and stores the result in `codeOutput`.

11. `int expectedOutput = 5;`

Statement in line 14 is the expected result of the `add(2, 3)` operation.

12. `try {`

Statement in line 16 starts a try-catch block — useful for **gracefully handling assertion failures** and printing messages.

13. `assertEquals("Addition test failed:", expectedOutput, codeOutput);`

Statement in line 17, compares `expectedOutput` with `codeOutput`.
If they are not equal, the assertion will fail and throw an `AssertionError` with the message `"Addition test failed:"`.

14. `} catch (AssertionError ae) {`

Statement in line 18, catches the `AssertionError` if the test fails. This prevents the program from crashing and lets you log the error.

2.2 Answer Sample

Based on the test code CalculatorTest.java, we can create the source code with the following steps:

1. `public class Calculator {`

In the test case, an object is created using `new Calculator()`. Therefore, we need to define a class named `Calculator` to match that.

2. `public int add(int a, int b) {`

   The test calls a method named `add`, which takes **two integer parameters**. So, the class must include a method named `add` with this signature: `int add(int, int)`.

3. `return a + b;`

   The test case expects the result of add(2, 3) to be 5. That means the method should return the sum of the two inputs a and b.

Source code which is made based on test code
Calculator.java

```
1  package dataStructure;
2
3  public class Calculator {
4      public int add(int a, int b) {
5          return a + b;
6      }
7  }
```

3. Problems

There are 12 problems to be solved with details explained in this section. All test cases are also available at the following link https://github.com/mustmentari/test-cases-ASD- .

3.1 Bubble Sort

Please create source code based on the test code below.

There are some additional explanations regarding the sorting algorithm as follows:

1. `import java.util.Arrays;`

   This import is **necessary** because we are comparing two arrays.

2. `int codeInput2 = 2;`

   We're testing **sorting logic with a limited number of iterations**. This allows the test case to check not just the final sorted array, but also **intermediate states** of the sorting process — for example, after 2 iterations only.

3. `int[] codeOutput = bSort.bubbleSort(codeInput1, codeInput2);`

   Calls the sorting method and returns the result **after a limited number of passes(codeInput2)**. bSort is an instance of the BubbleSort class, which implements the method bubbleSort(int[] arr, int maxIterations).

4.
```
assertEquals("Test 2:", Arrays.toString(expectedOutput),
Arrays.toString(codeOutput));
```

That compares **two arrays**, but converted to **Strings** first.

Test code for BubbleSort class

BubbleSortTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import java.util.Arrays;
 5  import org.junit.Test;
 6
 7  public class BubbleSortTest {
 8      // Intermediate state testing
 9      @Test
10      public void testBubbleSort1() {
11          BubbleSort bSort = new BubbleSort();
12          int[] codeInput1 = { 5, 2, 8, 1, 9 };
13          int codeInput2 = 2;
14          int[] codeOutput = bSort.bubbleSort(codeInput1, codeInput2);
15          // Expected state after 2 iterations
16          int[] expOutput = { 2, 1, 5, 8, 9 };
17
18          try {
19              assertEquals("Test 1:", Arrays.toString(expOutput), Arrays.toString(codeOutput));
20          } catch (AssertionError ae) {
21              System.out.println(ae);
21          }
22      }
23
24      @Test
25      public void testBubbleSort2() {
26          BubbleSort bSort = new BubbleSort();
27          int[] codeInput1 = { 121, -1, 9, 83, 52, -7, 3, -83, 75 };
28          int codeInput2 = 120;
29          int[] codeOutput = bSort.bubbleSort(codeInput1, codeInput2);
30          // Expected state after 120 iterations
31          int[] expOutput = { -83, -7, -1, 3, 9, 52, 75, 83, 121 };
32
33          try {
34              assertEquals("Test 2:", Arrays.toString(expOutput), Arrays.toString(codeOutput));
35          } catch (AssertionError ae) {
36              System.out.println(ae);
37          }
38      }
39
40      @Test
41      public void testBubbleSort3() {
42          BubbleSort bSort = new BubbleSort();
43          int[] codeInput1 = { -123, 53, 5, 23, -2, 48, 90, -83 };
44          int codeInput2 = -10;
45          int[] codeOutput = bSort.bubbleSort(codeInput1, codeInput2);
46          // Expected state after -10 iterations
47          int[] expOutput = { -123, 53, 5, 23, -2, 48, 90, -83 };
48
49          try {
50              assertEquals("Test 3:", Arrays.toString(expOutput), Arrays.toString(codeOutput));
51          } catch (AssertionError ae) {
52              System.out.println(ae);
53          }
54      }
55
56  }
```

3.2 Selection Sort

Please create source code based on the test code below.

Test code for SelectionSort class

SelectionSortTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import java.util.Arrays;
 5  import org.junit.Test;
 6
 7  public class SelectionSortTest {
 8      // Intermediate state testing
 9      @Test
10      public void testSelectionSort1() {
11          SelectionSort sSort = new SelectionSort();
12          int[] codeInput1 = { 50, -23, 9, 33, -101, 89 };
13          int codeInput2 = 4;
14          int[] codeOutput = sSort.selectionSort(codeInput1, codeInput2);
15          int[] expectedOutput = { -101, -23, 9, 33, 50, 89 };
16
17          try {
18              assertEquals("Test 1:", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
19          } catch (AssertionError ae) {
20              System.out.println(ae);
21          }
21      }
22
23      @Test
24      public void testSelectionSort2() {
25          SelectionSort sSort = new SelectionSort();
26          int[] codeInput1 = { 73, 3, -99, 47, 0, 890, -3, -56 };
27          int codeInput2 = 47;
28          int[] codeOutput = sSort.selectionSort(codeInput1, codeInput2);
29          int[] expectedOutput = { -99, -56, -3, 0, 3, 47, 73, 890 };
30
31          try {
32              assertEquals("Test 2:", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
33          } catch (AssertionError ae) {
34              System.out.println(ae);
35          }
36      }
37
38      @Test
39      public void testSelectionSort3() {
40          SelectionSort sSort = new SelectionSort();
41          int[] codeInput1 = { 65, -3, 0, 120, 23, 8, 90, -7, -22, 10 };
42          int codeInput2 = -50;
43          int[] codeOutput = sSort.selectionSort(codeInput1, codeInput2);
44          int[] expectedOutput = { 65, -3, 0, 120, 23, 8, 90, -7, -22, 10 };
45
46          try {
47              assertEquals("Test 3:", Arrays.toString(expectedOutput),
48  Arrays.toString(codeOutput));
          } catch (AssertionError ae) {
49              System.out.println(ae);
50          }
51      }
52  }
```

3.3 Insertion Sort

Please create source code based on the test code below.

Test code for InsertionSort class

InsertionSortTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import java.util.Arrays;
 5  import org.junit.Test;
 6
 7  public class InsertionSortTest {
 8      // Intermediate state testing
 9      @Test
10      public void testInsertionSort1() {
11          InsertionSort iSort = new InsertionSort();
12          int[] codeInput1 = { -85, 44, 3, 77, -231 };
13          int codeInput2 = 3;
14          int[] codeOutput = iSort.insertionSort(codeInput1, codeInput2);
15          int[] expectedOutput = { -85, 3, 44, 77, -231 };
16
17          try {
18              assertEquals("Test 1:", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
19          } catch (AssertionError ae) {
20              System.out.println(ae);
21          }
21      }
22
23      @Test
24      public void testInsertionSort2() {
25          InsertionSort iSort = new InsertionSort();
26          int[] codeInput1 = { 66, -1, 0, 104, -22, 8, 155, -7 };
27          int codeInput2 = 300;
28          int[] codeOutput = iSort.insertionSort(codeInput1, codeInput2);
29          int[] expectedOutput = { -22, -7, -1, 0, 8, 66, 104, 155 };
30
31          try {
32              assertEquals("Test 2:", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
33          } catch (AssertionError ae) {
34              System.out.println(ae);
35          }
36      }
37
38      @Test
39      public void testInsertionSort3() {
40          InsertionSort iSort = new InsertionSort();
41          int[] codeInput1 = { -7, 8, 99, -14, 89, 23, -123, 76, 33, -45 };
42          int codeInput2 = -8;
43          int[] codeOutput = iSort.insertionSort(codeInput1, codeInput2);
44          int[] expectedOutput = { -7, 8, 99, -14, 89, 23, -123, 76, 33, -45 };
45
46          try {
47              assertEquals("Test 3:", Arrays.toString(expectedOutput),
48  Arrays.toString(codeOutput));
          } catch (AssertionError ae) {
49              System.out.println(ae);
50          }
51      }
52  }
```

3.4 Quick Sort

Please create source code based on the test code below.

There are some additional explanations regarding the quick sort algorithm as follows:

1. `String codeInput3 = "middle";`

   This input parameter defines the pivot strategy used in the QuickSort algorithm. It can be `"middle"`, `"random"`, or `"last"`.

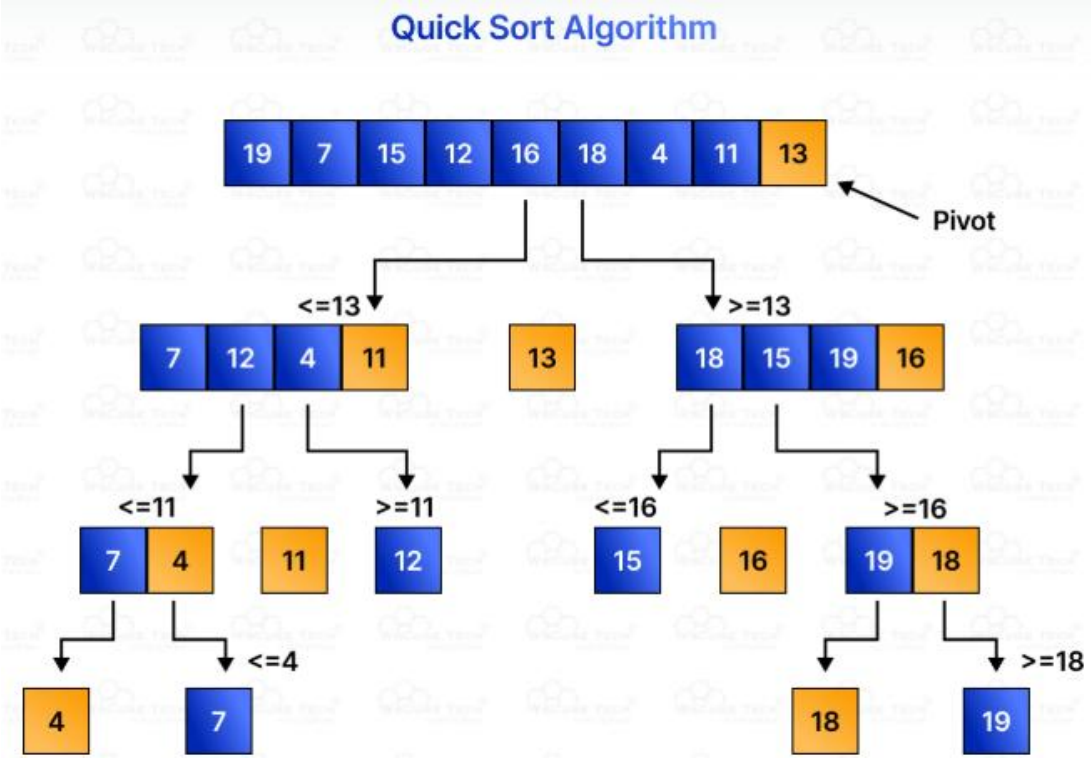
2. `int[] codeOutput = QuickSort.quickSort(codeInput1, codeInput2, codeInput3);`

   Calls the updated QuickSort method that now accepts three parameters: the array to be sorted (`codeInput1`), the maximum number of iterations (`codeInput2`), and the pivot strategy (`codeInput3`).

3. `assertEquals(Arrays.toString(expectedOutput), Arrays.toString(codeOutput));`

   Validates that the output of the sorting algorithm matches the expected result.

Quick Sort Algorithm illustration :



(Source : https://www.wscubetech.com/resources/dsa/quick-sort)

Test code for QuickSort class

QuickSortTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import org.junit.Test;
 5  import java.util.Arrays;
 6
 7  public class QuickSortTest {
 8
 9      @Test
10      // Middle Pivot
11      public void testQuickSort1() {
12          int[] codeInput1 = { 89, 23, -90, 0, 117, 90, 2, 1, 8, -9 };
13          int codeInput2 = 7;
14          String codeInput3 = "middle";
15
16          int[] codeOutput = QuickSort.quickSort(Arrays.copyOf(codeInput1, codeInput1.length),
    codeInput2, codeInput3);
17          int[] expectedOutput = { -90, -9, 0, 1, 2, 8, 23, 89, 90, 117 };
18
19          try {
20              assertEquals("Middle pivot sort failed", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
21          } catch (AssertionError ae) {
21              System.out.println(ae);
22          }
23      }
24
25      @Test
26      // Random Pivot
27      public void testQuickSort2() {
28          int[] codeInput1 = { -8, 0, 90, -12, 100, 23 };
29          int codeInput2 = 10;
30          String codeInput3 = "random";
31
32          int[] codeOutput = QuickSort.quickSort(Arrays.copyOf(codeInput1, codeInput1.length),
    codeInput2, codeInput3);
33          int[] expectedOutput = Arrays.copyOf(codeInput1, codeInput1.length);
34          Arrays.sort(expectedOutput);
35
36          try {
37              assertEquals("Random pivot sort failed", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
38          } catch (AssertionError ae) {
39              System.out.println(ae);
40          }
41      }
42
43      @Test
44      // Invalid Iterations
45      public void testQuickSort3() {
46          int[] codeInput1 = { 90, -2, -11, 0, 23, 78, 9, 102, -4, 89, -9 };
47          int codeInput2 = -3;
48          String codeInput3 = "last";
49
50          int[] codeOutput = QuickSort.quickSort(Arrays.copyOf(codeInput1, codeInput1.length),
51  codeInput2, codeInput3);
            int[] expectedOutput = Arrays.copyOf(codeInput1, codeInput1.length);
52
53          try {
54              assertEquals("Invalid iteration case failed", Arrays.toString(expectedOutput),
    Arrays.toString(codeOutput));
55          } catch (AssertionError ae) {
56              System.out.println(ae);
57          }
58      }
59  }
```

3.5 Shell Sort

Please create source code based on the test code below.

Test code for ShellSort class

ShellSortTest.java

```
1   package dataStructure;
2
3   import static org.junit.Assert.*;
4   import java.util.Arrays;
5   import org.junit.Test;
6
7   public class ShellSortTest {
8       // Intermediate state testing
9       @Test
10      public void testShellSort1() {
11          ShellSort sSort = new ShellSort();
12          int[] codeInput1 = { -9, -99, 2, 78, 45, -105, 78, -33 };
13          int codeInput2 = 3;
14
15          int[] codeOutput = sSort.shellSort(codeInput1, codeInput2); // Sorting result
16          // Expected intermediate state
17          int[] expectedOutput = { -9, -33, -99, 2, 45, 78, -105, 78 };
18
19          try {
20              assertEquals("Test 1:", Arrays.toString(codeOutput),
    Arrays.toString(expectedOutput));
21          } catch (AssertionError ae) {
21              System.out.println(ae);
22          }
23      }
24
25      @Test
26      public void testShellSort2() {
27          ShellSort sSort = new ShellSort();
28          int[] codeInput1 = { 67, -45, 0, -22, 0, 123, -3, -2, -34, 89, 2 };
29          int codeInput2 = 312;
30
31          int[] codeOutput = sSort.shellSort(codeInput1, codeInput2); // Sorting result
32          // Expected intermediate state
33          int[] expectedOutput = { -45, -34, -22, -3, -2, 0, 0, 2, 67, 89, 123 };
34
35          try {
36              assertEquals("Test 2:", Arrays.toString(codeOutput),
    Arrays.toString(expectedOutput));
37          } catch (AssertionError ae) {
38              System.out.println(ae);
39          }
40      }
41
42      @Test
43      public void testShellSort3() {
44          ShellSort sSort = new ShellSort();
45          int[] codeInput1 = { 8, -2, 89, -5, 47, 231, -9 };
46          int codeInput2 = -1;
47
48          int[] codeOutput = sSort.shellSort(codeInput1, codeInput2); // Sorting result
49          int[] expectedOutput = { 8, -2, 89, -5, 47, 231, -9 }; // Expected intermediate state
50
51          try {
52              assertEquals("Test 3:", Arrays.toString(codeOutput),
    Arrays.toString(expectedOutput));
53          } catch (AssertionError ae) {
54              System.out.println(ae);
55          }
56      }
57  }
```

3.6 Sequential Search

Please create source code based on the test code below.

There are some additional explanations regarding the searching algorithm as follows:

1. `int target = 317;`

   Statements in lines 14, 30 and 49 mention the variable `target` represent the value to find in the element array.

2. `int[] expectedOutput = { 8, 9 }; // Found at index 8 after 9 iterations`
   `int[] expectedOutput = { -1, 10 }; // Not found after 10 checks`

   Statements in lines 16, 32, and 51 mention the unique statement to search tests. It not only checks whether the item was found, but also **how efficiently** (how many steps).

Test code for SequentialSearch class

SequentialSearchTest.java

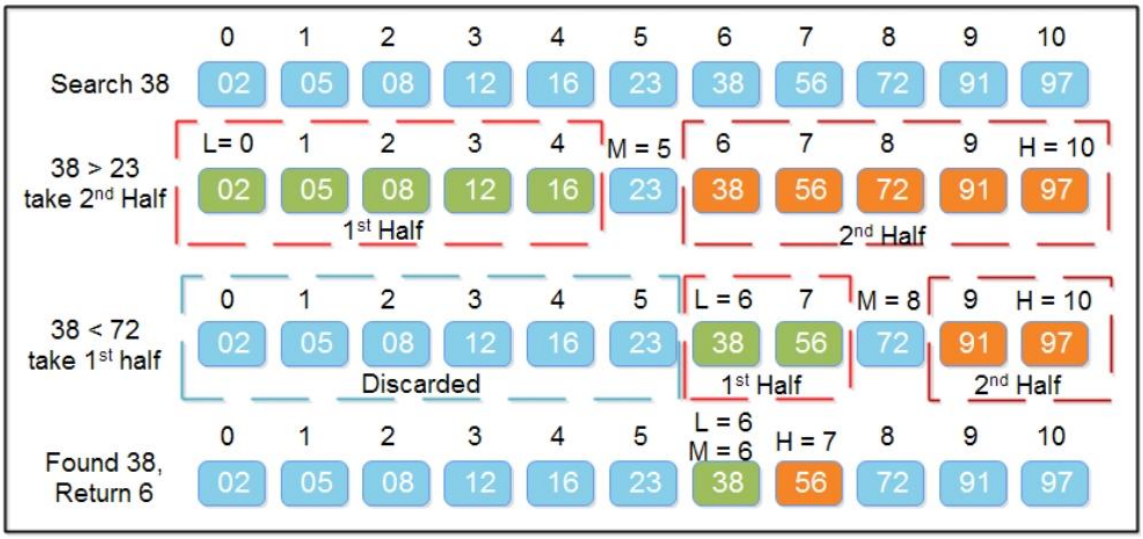
```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import java.util.Arrays;
 5  import org.junit.Test;
 6
 7  public class SequentialSearchTest {
 8      @Test
 9      public void testSequentialSearch1() {
10          SequentialSearch sSearch = new SequentialSearch();
11
12          int[] codeInput = { -1, 2, 15, 27, 90, 110, 113, 215, 317, 320 }; // Array to search
13          int target = 317; // Target value to find
14          int[] codeOutput = sSearch.sequentialSearch(codeInput, target);
15          int[] expectedOutput = { 8, 9 };
16          try {
17              assertEquals("Test 1: Sequential Search with target 317",
18  Arrays.toString(expectedOutput),
                        Arrays.toString(codeOutput));
19          } catch (AssertionError ae) {
20              System.out.println(ae);
21          }
21      }
22
23      @Test
24      public void testSequentialSearch2() {
25          SequentialSearch sSearch = new SequentialSearch();
26
27          int[] codeInput = { -1, 2, 15, 27, 90, 110, 113, 215, 317, 320 }; // Array to search
28          int target = 15;
29          int[] codeOutput = sSearch.sequentialSearch(codeInput, target);
30          int[] expectedOutput = { 2, 3 };
31
32          try {
33              assertEquals("Test 1: Sequential Search with target 15",
34  Arrays.toString(expectedOutput),
                        Arrays.toString(codeOutput));
35          } catch (AssertionError ae) {
36              System.out.println(ae);
37          }
38      }
39
```

```
40      // Test for target not found
41      @Test
42      public void testSequentialSearch3() {
43          SequentialSearch sSearch = new SequentialSearch();
44
45          int[] codeInput = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 }; // Array to search
46          int target = 321;
47          int[] codeOutput = sSearch.sequentialSearch(codeInput, target);
48          int[] expectedOutput = { -1, 10 }; // Expected index (-1) and iterations (10)
49
50          try {
51              assertEquals("Test3: Sequential Search with target 321 (not found)",
    Arrays.toString(expectedOutput),
52                      Arrays.toString(codeOutput));
53          } catch (AssertionError ae) {
54              System.out.println(ae);
55          }
56      }
57  }
```

3.7 Binary Search

Please create source code based on the test code below.

Binary Search Illustration :



(Source : https://ecomputernotes.com/what-is-c/binary-search-in-c)

Test code for BinarySearch class

BinarySearchTest.java

```
1   package dataStructure;
2
3   import static org.junit.Assert.*;
4   import java.util.Arrays;
5   import org.junit.Test;
6
7   public class BinarySearchTest {
8       @Test
9       public void testBinarySearch1() {
10          // Initialize BinarySearch object
11          BinarySearch bSearch = new BinarySearch();
12
13          // Input for the test case: array and target
```

```
14          int[] codeInput = { -3, -1, 4, 7, 45, 51, 52, 70, 75, 90 }; // Array to search
15          int target = 7; // Target value to find
16
17          // Call the binarySearch method
18          int[] codeOutput = bSearch.binarySearch(codeInput, target);
19
20          // Expected output: index of the target and the number of iterations it took
21          int[] expectedOutput = { 3, 4 };
22
23          try {
                assertEquals("Test 1: Binary Search with target 7",
    Arrays.toString(expectedOutput),
24                      Arrays.toString(codeOutput));
25          } catch (AssertionError ae) {
26              System.out.println(ae);
27          }
28      }
29
30      @Test
31      public void testBinarySearch2() {
32          // Initialize BinarySearch object
33          BinarySearch bSearch = new BinarySearch();
34
35          // Input for the test case: array and target
36          int[] codeInput = { -3, -1, 4, 7, 45, 51, 52, 70, 75, 90 }; // Array to search
37          int target = 75; // Target value to find
38
39          // Call the binarySearch method
40          int[] codeOutput = bSearch.binarySearch(codeInput, target);
41
42          // Expected output: index of the target and the number of iterations it took
43          int[] expectedOutput = { 8, 3 };
44
45          try {
46              assertEquals("Test 2: Binary Search with target 17",
    Arrays.toString(expectedOutput),
47                      Arrays.toString(codeOutput));
48          } catch (AssertionError ae) {
49              System.out.println(ae);
50          }
51      }
52
53      @Test
54      public void testBinarySearch3() {
55          // Initialize BinarySearch object
56          BinarySearch bSearch = new BinarySearch();
57
58          // Input for the test case: array and target
59          int[] codeInput = { -3, -1, 4, 7, 45, 51, 52, 70, 75, 90 }; // Array to search
60          int target = 100; // Target value that is not in the array
61
62          // Call the binarySearch method
63          int[] codeOutput = bSearch.binarySearch(codeInput, target);
64
65          // Expected output: -1 (not found) and the number of iterations it took
66          int[] expectedOutput = { -1, 4 }; // Expected index (-1) and iterations (4)
67
68          try {
69              assertEquals("Test 3: Binary Search with target 100 (not found)",
    Arrays.toString(expectedOutput),
70                      Arrays.toString(codeOutput));
71          } catch (AssertionError ae) {
72              System.out.println(ae);
73          }
74      }
75  }
```

3.8 Stack

Please create source code based on the test code below.

There are some additional explanations regarding the stack algorithm as follows:

1. `stack.isEmpty();`

   To verify whether the stack is empty after an operation

2. `stack.isFull();`

   To test overflow conditions

Test code for Stack class

StackTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import org.junit.Test;
 5
 6  public class StackTest {
 7
 8      @Test
 9      // push-pop-push-push
10      public void testStack1() {
11          Stack stack = new Stack(10);
12
13          // Input values
14          int push1 = 99;
15          int push2 = 50;
16          int push3 = -34;
17
18          // Expected and actual outputs
            int expectedOutput1 = 99; // peek after 1st push
19          boolean expectedOutput2 = false; // isEmpty after 1st push
20          int codeOutput1 = 0;
21          boolean codeOutput2 = true;
21
22          int expectedOutput3 = 0; // peek after pop (stack empty)
23          boolean expectedOutput4 = true; // isEmpty after pop
24          int codeOutput3 = 0;
25          boolean codeOutput4 = false;
26
27          int expectedOutput5 = 50; // peek after 2nd push
28          boolean expectedOutput6 = false; // isEmpty after 2nd push
29          int codeOutput5 = 0;
30          boolean codeOutput6 = true;
31
32          int expectedOutput7 = -34; // peek after 3rd push
33          boolean expectedOutput8 = false; // isEmpty after 3rd push
34          int codeOutput7 = 0;
35          boolean codeOutput8 = true;
36
37          try {
38              // Step 1: Push(99)
39              stack.push(push1);
40              codeOutput1 = stack.peek();
41              codeOutput2 = stack.isEmpty();
42
43              assertEquals("Peek after first push", expectedOutput1, codeOutput1);
44              assertEquals("isEmpty after first push", expectedOutput2, codeOutput2);
45
46              // Step 2: Pop()
47              stack.pop();
```

```
48              codeOutput3 = stack.isEmpty() ? 0 : stack.peek(); // Peek if not empty
                codeOutput4 = stack.isEmpty();
49
50              assertEquals("Peek after pop", expectedOutput3, codeOutput3);
51              assertEquals("isEmpty after pop", expectedOutput4, codeOutput4);
52
53              // Step 3: Push(50)
54              stack.push(push2);
55              codeOutput5 = stack.peek();
56              codeOutput6 = stack.isEmpty();
57
58              assertEquals("Peek after second push", expectedOutput5, codeOutput5);
59              assertEquals("isEmpty after second push", expectedOutput6, codeOutput6);
60
61              // Step 4: Push(-34)
62              stack.push(push3);
63              codeOutput7 = stack.peek();
64              codeOutput8 = stack.isEmpty();
65
66              assertEquals("Peek after third push", expectedOutput7, codeOutput7);
67              assertEquals("isEmpty after third push", expectedOutput8, codeOutput8);
68
69          } catch (AssertionError ae) {
70              System.out.println(ae);
71          }
72      }
73
74      @Test
75      // pop-push-push-pop
76      public void testStack2() {
77          Stack stack = new Stack(10);
78
79          // Input values
80          int push1 = 77;
81          int push2 = -12;
82
83          // Expected and actual outputs
84          int expectedOutput1 = 0; // peek after pop on empty (assume 0 if empty)
85          boolean expectedOutput2 = true; // isEmpty should still be true
86          int codeOutput1 = 0;
87          boolean codeOutput2 = false;
88
89          int expectedOutput3 = 77; // peek after first push
90          boolean expectedOutput4 = false; // isEmpty should be false
91          int codeOutput3 = 0;
92          boolean codeOutput4 = true;
93
94          int expectedOutput5 = -12; // peek after second push
95          boolean expectedOutput6 = false;
96          int codeOutput5 = 0;
97          boolean codeOutput6 = true;
98
99          int expectedOutput7 = 77; // peek after final pop (should leave 77 on top)
100         boolean expectedOutput8 = false;
102         int codeOutput7 = 0;
103         boolean codeOutput8 = true;
104
105         try {
106             // Step 1: Pop (from empty stack)
107             stack.pop(); // should not crash if stack handles it gracefully
108             codeOutput1 = stack.isEmpty() ? 0 : stack.peek(); // avoid peek if empty
109             codeOutput2 = stack.isEmpty();
110
111             assertEquals("Peek after pop on empty stack", expectedOutput1, codeOutput1);
112             assertEquals("isEmpty after pop on empty stack", expectedOutput2, codeOutput2);
113
114             // Step 2: Push(77)
115             stack.push(push1);
116             codeOutput3 = stack.peek();
117             codeOutput4 = stack.isEmpty();
118
```

```
119             assertEquals("Peek after first push", expectedOutput3, codeOutput3);
120             assertEquals("isEmpty after first push", expectedOutput4, codeOutput4);
121
122             // Step 3: Push(-12)
123             stack.push(push2);
124             codeOutput5 = stack.peek();
125             codeOutput6 = stack.isEmpty();
126
127             assertEquals("Peek after second push", expectedOutput5, codeOutput5);
128             assertEquals("isEmpty after second push", expectedOutput6, codeOutput6);
129
130             // Step 4: Pop (should remove -12)
131             stack.pop();
132             codeOutput7 = stack.peek();
133             codeOutput8 = stack.isEmpty();
134
135             assertEquals("Peek after final pop", expectedOutput7, codeOutput7);
136             assertEquals("isEmpty after final pop", expectedOutput8, codeOutput8);
137
138         } catch (AssertionError ae) {
139             System.out.println(ae);
140         }
        }
141
142     @Test
143     // push-pop-push until overflow
144     public void testStack3() {
145         Stack stack = new Stack(3);
146
147         // Input
148         int codeInput1 = 10;
149         int codeInput2 = 20;
150         int codeInput3 = 30;
151         int codeInput4 = 40;
152         int codeInput5 = 50; // overflow testing 3
153
154         // Expected and actual outputs
155         int expectedOutput1 = 10; // Peek after push(10)
156         boolean expectedOutput2 = false; // isEmpty after push(10)
157         int codeOutput1 = 0;
158         boolean codeOutput2 = true;
159
160         int expectedOutput3 = 20; // Peek after push(20)
161         boolean expectedOutput4 = false; // isEmpty after push(20)
162         int codeOutput3 = 0;
163         boolean codeOutput4 = true;
164
165         int expectedOutput5 = 10; // Peek after pop()
166         boolean expectedOutput6 = false;
167         int codeOutput5 = 0;
168         boolean codeOutput6 = true;
169
170         int expectedOutput7 = 30; // Peek after push(30)
171         boolean expectedOutput8 = false;
172         int codeOutput7 = 0;
173         boolean codeOutput8 = true;
174
175         int expectedOutput9 = 40; // Peek after push(40)
176         boolean expectedOutput10 = false;
177         int codeOutput9 = 0;
178         boolean codeOutput10 = true;
179
180         int expectedOutput11 = 50; // Peek after push(50)
181         boolean expectedOutput12 = true; // isFull after push(50), overflow
182         int codeOutput11 = 0;
183         boolean codeOutput12 = false;
184
185         try {
186             // Step 1: push(10)
187             stack.push(codeInput1);
188             codeOutput1 = stack.peek();
```

```
189             codeOutput2 = stack.isEmpty();
190             assertEquals("Peek after push(10)", expectedOutput1, codeOutput1);
191             assertEquals("isEmpty after push(10)", expectedOutput2, codeOutput2);
192
193             // Step 2: push(20)
194             stack.push(codeInput2);
195             codeOutput3 = stack.peek();
196             codeOutput4 = stack.isEmpty();
197             assertEquals("Peek after push(20)", expectedOutput3, codeOutput3);
198             assertEquals("isEmpty after push(20)", expectedOutput4, codeOutput4);
199
200             // Step 3: pop()
201             stack.pop();
202             codeOutput5 = stack.peek();
203             codeOutput6 = stack.isEmpty();
204             assertEquals("Peek after pop()", expectedOutput5, codeOutput5);
205             assertEquals("isEmpty after pop()", expectedOutput6, codeOutput6);
206
207             // Step 4: push(30)
208             stack.push(codeInput3);
209             codeOutput7 = stack.peek();
210             codeOutput8 = stack.isEmpty();
211             assertEquals("Peek after push(30)", expectedOutput7, codeOutput7);
212             assertEquals("isEmpty after push(30)", expectedOutput8, codeOutput8);
213
214             // Step 5: push(40)
215             stack.push(codeInput4);
216             codeOutput9 = stack.peek();
217             codeOutput10 = stack.isEmpty();
218             assertEquals("Peek after push(40)", expectedOutput9, codeOutput9);
219             assertEquals("isEmpty after push(40)", expectedOutput10, codeOutput10);
220
221             // Step 6: push(50) → overflow check
222             if (!stack.isFull()) {
223                 stack.push(codeInput5);
224             }
225             codeOutput11 = stack.peek();
226             codeOutput12 = stack.isFull();
227
228             assertEquals("Peek after push(50)", expectedOutput11, codeOutput11);
229             assertEquals("isFull after push(50)", expectedOutput12, codeOutput12);
230
231         } catch (AssertionError ae) {
232             System.out.println(ae);
233         }
234     }
235 }
```

3.9 Queue

Please create source code based on the test code below.

Test code for Queue class

QueueTest.java

```
1   package dataStructure;
2
3   import static org.junit.Assert.*;
4   import org.junit.Test;
5
6   public class QueueTest {
7
8       @Test
9       // enqueue-dequeue-enqueue-enqueue
10      public void testQueue1() {
11          Queue queue = new Queue(10);
12
```

```
13          // Input values
14          int enqueue1 = 99;
15          int enqueue2 = 50;
16          int enqueue3 = -34;
17
18          // Expected and actual outputs
19          int expectedOutput1 = 99; // peek after 1st enqueue
20          boolean expectedOutput2 = false; // isEmpty after 1st enqueue
21          int codeOutput1 = 0;
21          boolean codeOutput2 = true;
22
23          int expectedOutput3 = 0; // peek after dequeue (queue empty)
24          boolean expectedOutput4 = true; // isEmpty after dequeue
25          int codeOutput3 = 0;
26          boolean codeOutput4 = false;
27
28          int expectedOutput5 = 50; // peek after 2nd enqueue
29          boolean expectedOutput6 = false; // isEmpty after 2nd enqueue
30          int codeOutput5 = 0;
31          boolean codeOutput6 = true;
32
33          int expectedOutput7 = -34; // peek after 3rd enqueue
34          boolean expectedOutput8 = false; // isEmpty after 3rd enqueue
35          int codeOutput7 = 0;
36          boolean codeOutput8 = true;
37
38          try {
39              // Step 1: Enqueue(99)
40              queue.enqueue(enqueue1);
41              codeOutput1 = queue.peek();
42              codeOutput2 = queue.isEmpty();
43
44              assertEquals("Peek after first enqueue", expectedOutput1, codeOutput1);
45              assertEquals("isEmpty after first enqueue", expectedOutput2, codeOutput2);
46
47              // Step 2: Dequeue()
48              queue.dequeue();
49              codeOutput3 = queue.isEmpty() ? 0 : queue.peek(); // Peek if not empty
50              codeOutput4 = queue.isEmpty();
51
52              assertEquals("Peek after dequeue", expectedOutput3, codeOutput3);
53              assertEquals("isEmpty after dequeue", expectedOutput4, codeOutput4);
54
55              // Step 3: Enqueue(50)
56              queue.enqueue(enqueue2);
57              codeOutput5 = queue.peek();
58              codeOutput6 = queue.isEmpty();
59
60              assertEquals("Peek after second enqueue", expectedOutput5, codeOutput5);
61              assertEquals("isEmpty after second enqueue", expectedOutput6, codeOutput6);
62
63              // Step 4: Enqueue(-34)
64              queue.enqueue(enqueue3);
65              codeOutput7 = queue.peek();
66              codeOutput8 = queue.isEmpty();
67
68              assertEquals("Peek after third enqueue", expectedOutput7, codeOutput7);
69              assertEquals("isEmpty after third enqueue", expectedOutput8, codeOutput8);
70
71          } catch (AssertionError ae) {
72              System.out.println(ae);
73          }
74      }
75
76      @Test
77      // dequeue-enqueue-enquque-dequeue
78      public void testQueue2() {
79          Queue queue = new Queue(10);
80
81          // Input values
82          int enqueue1 = 77;
```

```
 83          int enqueue2 = -12;
 84
 85          // Expected and actual outputs
 86          int expectedOutput1 = 0; // peek after dequeue on empty (assume 0 if empty)
 87          boolean expectedOutput2 = true; // isEmpty should still be true
 88          int codeOutput1 = 0;
 89          boolean codeOutput2 = false;
 90
 91          int expectedOutput3 = 77; // peek after first enqueue
 92          boolean expectedOutput4 = false; // isEmpty should be false
 93          int codeOutput3 = 0;
 94          boolean codeOutput4 = true;
 95
 96          int expectedOutput5 = -12; // peek after second enqueue
 97          boolean expectedOutput6 = false;
 98          int codeOutput5 = 0;
 99          boolean codeOutput6 = true;
100
102          int expectedOutput7 = 77; // peek after final dequeue (should leave 77 in front)
103          boolean expectedOutput8 = false;
104          int codeOutput7 = 0;
105          boolean codeOutput8 = true;
106
107          try {
108              // Step 1: Dequeue (from empty queue)
109              queue.dequeue(); // should not crash if queue handles it gracefully
110              codeOutput1 = queue.isEmpty() ? 0 : queue.peek(); // avoid peek if empty
111              codeOutput2 = queue.isEmpty();
112
113              assertEquals("Peek after dequeue on empty queue", expectedOutput1, codeOutput1);
114              assertEquals("isEmpty after dequeue on empty queue", expectedOutput2, codeOutput2);
115
116              // Step 2: Enqueue(77)
117              queue.enqueue(enqueue1);
118              codeOutput3 = queue.peek();
119              codeOutput4 = queue.isEmpty();
120
121              assertEquals("Peek after first enqueue", expectedOutput3, codeOutput3);
122              assertEquals("isEmpty after first enqueue", expectedOutput4, codeOutput4);
123
124              // Step 3: Enqueue(-12)
125              queue.enqueue(enqueue2);
126              codeOutput5 = queue.peek();
127              codeOutput6 = queue.isEmpty();
128
129              assertEquals("Peek after second enqueue", expectedOutput5, codeOutput5);
130              assertEquals("isEmpty after second enqueue", expectedOutput6, codeOutput6);
131
132              // Step 4: Dequeue (should remove 77)
133              queue.dequeue();
134              codeOutput7 = queue.peek();
135              codeOutput8 = queue.isEmpty();
136
137              assertEquals("Peek after final dequeue", expectedOutput7, codeOutput7);
138              assertEquals("isEmpty after final dequeue", expectedOutput8, codeOutput8);
139
140          } catch (AssertionError ae) {
141              System.out.println(ae);
142          }
143      }
144
145      @Test
146      // enqueue-dequeue-enqueue-enqueue (overflow)
147      public void testQueue3() {
148          Queue queue = new Queue(3);
149
150          // Input values
151          int enqueue1 = 10;
152          int enqueue2 = 20;
153          int enqueue3 = 30;
154          int enqueue4 = 40;
```

```
155
156         // Expected and actual outputs
157         int expectedOutput1 = 10; // Peek after enqueue(10)
158         boolean expectedOutput2 = false; // isEmpty after enqueue(10)
159         int codeOutput1 = 0;
160         boolean codeOutput2 = true;
161
162         int expectedOutput3 = 20; // Peek after enqueue(20)
163         boolean expectedOutput4 = false; // isEmpty after enqueue(20)
164         int codeOutput3 = 0;
165         boolean codeOutput4 = true;
166
167         int expectedOutput5 = 10; // Peek after dequeue
168         boolean expectedOutput6 = false;
169         int codeOutput5 = 0;
170         boolean codeOutput6 = true;
171
172         int expectedOutput7 = 30; // Peek after enqueue(30)
173         boolean expectedOutput8 = false;
174         int codeOutput7 = 0;
175         boolean codeOutput8 = true;
176
177         int expectedOutput9 = 40; // Peek after enqueue(40)
178         boolean expectedOutput10 = false;
179         int codeOutput9 = 0;
180         boolean codeOutput10 = true;
181
182         try {
183             // Step 1: Enqueue(10)
184             queue.enqueue(enqueue1);
185             codeOutput1 = queue.peek();
186             codeOutput2 = queue.isEmpty();
187             assertEquals("Peek after enqueue(10)", expectedOutput1, codeOutput1);
188             assertEquals("isEmpty after enqueue(10)", expectedOutput2, codeOutput2);
189
190             // Step 2: Enqueue(20)
191             queue.enqueue(enqueue2);
192             codeOutput3 = queue.peek();
193             codeOutput4 = queue.isEmpty();
194             assertEquals("Peek after enqueue(20)", expectedOutput3, codeOutput3);
195             assertEquals("isEmpty after enqueue(20)", expectedOutput4, codeOutput4);
196
197             // Step 3: Dequeue
198             queue.dequeue();
199             codeOutput5 = queue.peek();
200             codeOutput6 = queue.isEmpty();
201             assertEquals("Peek after dequeue", expectedOutput5, codeOutput5);
202             assertEquals("isEmpty after dequeue", expectedOutput6, codeOutput6);
203
204             // Step 4: Enqueue(30)
205             queue.enqueue(enqueue3);
206             codeOutput7 = queue.peek();
207             codeOutput8 = queue.isEmpty();
208             assertEquals("Peek after enqueue(30)", expectedOutput7, codeOutput7);
209             assertEquals("isEmpty after enqueue(30)", expectedOutput8, codeOutput8);
210
211             // Step 5: Enqueue(40)
212             queue.enqueue(enqueue4);
213             codeOutput9 = queue.peek();
214             codeOutput10 = queue.isEmpty();
215             assertEquals("Peek after enqueue(40)", expectedOutput9, codeOutput9);
216             assertEquals("isEmpty after enqueue(40)", expectedOutput10, codeOutput10);
217
218         } catch (AssertionError ae) {
219             System.out.println(ae);
220         }
221     }
222 }
```

3.10 Single Linked List

Please create source code based on the test code below.

Test code for Single Linked List class

SingleLinkedListTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import org.junit.Test;
 5
 6  public class SingleLinkedListTest {
 7
 8      @Test
 9      // addFirst-RemoveFirst-addFirst-addFirst-removeFirst
10      public void testSLL1() {
11          SingleLinkedList list = new SingleLinkedList();
12
13          // Input values
14          int codeInput1 = 10;
15          int codeInput2 = 20;
16          int codeInput3 = 30;
17
18          // Expected and actual outputs
19          int expectedOutput1 = 10; // peek after first insert (head element)
20          boolean expectedOutput2 = false; // isEmpty after first insert
21          int codeOutput1 = 0;
21          boolean codeOutput2 = true;
22
23          int expectedOutput3 = 0; // peek after remove (list empty)
24          boolean expectedOutput4 = true; // isEmpty after remove
25          int codeOutput3 = 0;
26          boolean codeOutput4 = false;
27
28          int expectedOutput5 = 20; // peek after second insert (new head)
29          boolean expectedOutput6 = false; // isEmpty after second insert
30          int codeOutput5 = 0;
31          boolean codeOutput6 = true;
32
33          int expectedOutput7 = 30; // peek after third insert
34          boolean expectedOutput8 = false; // isEmpty after third insert
35          int codeOutput7 = 0;
36          boolean codeOutput8 = true;
37
38          try {
39              // Step 1: AddFirst(10)
40              list.addFirst(codeInput1);
41              codeOutput1 = list.peekFirst();
42              codeOutput2 = list.isEmpty();
43
44              assertEquals("Peek after first insert", expectedOutput1, codeOutput1);
45              assertEquals("isEmpty after first insert", expectedOutput2, codeOutput2);
46
47              // Step 2: RemoveFirst() (before remove, check if list is empty or not)
48              list.removeFirst();
49              codeOutput3 = list.isEmpty() ? 0 : list.peekFirst(); // Peek if not empty
50              codeOutput4 = list.isEmpty();
51
52              assertEquals("Peek after remove", expectedOutput3, codeOutput3);
53              assertEquals("isEmpty after remove", expectedOutput4, codeOutput4);
54
55              // Step 3: AddFirst(20)
56              list.addFirst(codeInput2);
57              codeOutput5 = list.peekFirst();
58              codeOutput6 = list.isEmpty();
59
60              assertEquals("Peek after second insert", expectedOutput5, codeOutput5);
61              assertEquals("isEmpty after second insert", expectedOutput6, codeOutput6);
62
```

```
            // Step 4: AddFirst(30)
            list.addFirst(codeInput3);
            codeOutput7 = list.peekFirst();
            codeOutput8 = list.isEmpty();

            assertEquals("Peek after third insert", expectedOutput7, codeOutput7);
            assertEquals("isEmpty after third insert", expectedOutput8, codeOutput8);

            // Step 5: RemoveFirst() again
            list.removeFirst();
            int expectedOutput9 = 20; // peek should show 20 after removal of 30
            boolean expectedOutput10 = false; // isEmpty should be false
            codeOutput7 = list.peekFirst();
            codeOutput8 = list.isEmpty();

            assertEquals("Peek after second remove", expectedOutput9, codeOutput7);
            assertEquals("isEmpty after second remove", expectedOutput10, codeOutput8);

        } catch (AssertionError ae) {
            System.out.println(ae);
        }
    }

    @Test
    // addLast-removeLast-addFirst-addFirst-removeLast
    public void testSLL2() {
        SingleLinkedList list = new SingleLinkedList();

        // Input values
        int codeInput1 = 10;
        int codeInput2 = 20;
        int codeInput3 = 30;

        // Expected and actual outputs
        int expectedOutput1 = 10; // peek after first insert (head element)
        boolean expectedOutput2 = false; // isEmpty after first insert
        int codeOutput1 = 0;
        boolean codeOutput2 = true;

        int expectedOutput3 = 0; // peek after remove (list empty)
        boolean expectedOutput4 = true; // isEmpty after remove
        int codeOutput3 = 0;
        boolean codeOutput4 = false;

        int expectedOutput5 = 20; // peek after second insert (new head)
        boolean expectedOutput6 = false; // isEmpty after second insert
        int codeOutput5 = 0;
        boolean codeOutput6 = true;

        int expectedOutput7 = 30; // peek after third insert
        boolean expectedOutput8 = false; // isEmpty after third insert
        int codeOutput7 = 0;
        boolean codeOutput8 = true;

        try {
            // Step 1: AddLast(10)
            list.addLast(codeInput1);
            codeOutput1 = list.peekFirst();
            codeOutput2 = list.isEmpty();

            assertEquals("Peek after first insert", expectedOutput1, codeOutput1);
            assertEquals("isEmpty after first insert", expectedOutput2, codeOutput2);

            // Step 2: RemoveLast() (before remove, check if list is empty or not)
            list.removeLast();
            codeOutput3 = list.isEmpty() ? 0 : list.peekFirst(); // Peek if not empty
            codeOutput4 = list.isEmpty();

            assertEquals("Peek after remove", expectedOutput3, codeOutput3);
            assertEquals("isEmpty after remove", expectedOutput4, codeOutput4);

```

```
135             // Step 3: AddFirst(20)
136             list.addFirst(codeInput2);
137             codeOutput5 = list.peekFirst();
138             codeOutput6 = list.isEmpty();
139 
140             assertEquals("Peek after second insert", expectedOutput5, codeOutput5);
141             assertEquals("isEmpty after second insert", expectedOutput6, codeOutput6);
142 
143             // Step 4: AddFirst(30)
144             list.addFirst(codeInput3);
145             codeOutput7 = list.peekFirst();
146             codeOutput8 = list.isEmpty();
147 
148             assertEquals("Peek after third insert", expectedOutput7, codeOutput7);
149             assertEquals("isEmpty after third insert", expectedOutput8, codeOutput8);
150 
151             // Step 5: RemoveLast() again
152             list.removeLast();
153             int expectedOutput9 = 20; // peek should show 20 after removal of 30
154             boolean expectedOutput10 = false; // isEmpty should be false
155             codeOutput7 = list.peekFirst();
156             codeOutput8 = list.isEmpty();
157 
158             assertEquals("Peek after second remove", expectedOutput9, codeOutput7);
159             assertEquals("isEmpty after second remove", expectedOutput10, codeOutput8);
160 
161         } catch (AssertionError ae) {
162             System.out.println(ae);
163         }
164     }
165 
166     @Test
167     // removeFirst-addFirst-addFirst-removeFirst-addLast-addLast-removeLast
168     public void testSLL3() {
169         SingleLinkedList list = new SingleLinkedList();
170 
171         // Input values
172         int codeInput1 = 10;
173         int codeInput2 = 20;
174         int codeInput3 = 30;
175         int codeInput4 = 40;
176 
177         // Expected and actual outputs
178         boolean expectedOutput1 = true; // isEmpty after removeFirst on empty list
179         boolean codeOutput1 = false;
180 
181         int expectedOutput2 = 10; // peek after addFirst(10)
182         boolean expectedOutput3 = false; // isEmpty after addFirst(10)
183         int codeOutput2 = 0;
184         boolean codeOutput3 = true;
185 
186         int expectedOutput4 = 20; // peek after addFirst(20)
187         boolean expectedOutput5 = false; // isEmpty after addFirst(20)
188         int codeOutput4 = 0;
189         boolean codeOutput5 = true;
190 
191         int expectedOutput6 = 10; // peek after removeFirst()
192         boolean expectedOutput7 = false; // still not empty
193         int codeOutput6 = 0;
194         boolean codeOutput7 = true;
195 
196         int expectedOutput8 = 10; // peekFirst still 10 after addLast(30)
197         boolean expectedOutput9 = false;
198         int codeOutput8 = 0;
199         boolean codeOutput9 = true;
200 
201         int expectedOutput10 = 10; // peekFirst still 10 after addLast(40)
202         boolean expectedOutput11 = false;
203         int codeOutput10 = 0;
204         boolean codeOutput11 = true;
205 
```

```
206         int expectedOutput12 = 10; // peekFirst still 10 after removeLast()
207         boolean expectedOutput13 = false;
208         int codeOutput12 = 0;
209         boolean codeOutput13 = true;
210 
211         try {
212             // Step 1: removeFirst() on empty list
213             list.removeFirst();
214             codeOutput1 = list.isEmpty();
215             assertEquals("isEmpty after removeFirst on empty", expectedOutput1, codeOutput1);
216 
217             // Step 2: addFirst(10)
218             list.addFirst(codeInput1);
219             codeOutput2 = list.peekFirst();
220             codeOutput3 = list.isEmpty();
221             assertEquals("Peek after addFirst(10)", expectedOutput2, codeOutput2);
222             assertEquals("isEmpty after addFirst(10)", expectedOutput3, codeOutput3);
223 
224             // Step 3: addFirst(20)
225             list.addFirst(codeInput2);
226             codeOutput4 = list.peekFirst();
227             codeOutput5 = list.isEmpty();
228             assertEquals("Peek after addFirst(20)", expectedOutput4, codeOutput4);
229             assertEquals("isEmpty after addFirst(20)", expectedOutput5, codeOutput5);
230 
231             // Step 4: removeFirst() → front should now be 10
232             list.removeFirst();
233             codeOutput6 = list.peekFirst();
234             codeOutput7 = list.isEmpty();
235             assertEquals("Peek after removeFirst()", expectedOutput6, codeOutput6);
236             assertEquals("isEmpty after removeFirst()", expectedOutput7, codeOutput7);
237 
238             // Step 5: addLast(30)
239             list.addLast(codeInput3);
240             codeOutput8 = list.peekFirst();
241             codeOutput9 = list.isEmpty();
242             assertEquals("Peek after addLast(30)", expectedOutput8, codeOutput8);
243             assertEquals("isEmpty after addLast(30)", expectedOutput9, codeOutput9);
244 
245             // Step 6: addLast(40)
246             list.addLast(codeInput4);
247             codeOutput10 = list.peekFirst();
248             codeOutput11 = list.isEmpty();
249             assertEquals("Peek after addLast(40)", expectedOutput10, codeOutput10);
250             assertEquals("isEmpty after addLast(40)", expectedOutput11, codeOutput11);
251 
252             // Step 7: removeLast()
253             list.removeLast();
254             codeOutput12 = list.peekFirst();
255             codeOutput13 = list.isEmpty();
256             assertEquals("Peek after removeLast()", expectedOutput12, codeOutput12);
257             assertEquals("isEmpty after removeLast()", expectedOutput13, codeOutput13);
258 
259         } catch (AssertionError ae) {
260             System.out.println("Test failed: " + ae.getMessage());
261         }
262     }
263 }
```

3.11 Double Linked List

Please create source code based on the test code below.

Test code for Double Linked List class

DoubleLinkedListTest.java

```
1   package dataStructure;
```

```
 2
 3  import static org.junit.Assert.*;
 4  import org.junit.Test;
 5
 6  public class DoubleLinkedListTest {
 7
 8      @Test
 9      // addFirst-removeFirst-addFirst-addFirst-removeFirst
10      public void testDLL1() {
11          DoubleLinkedList list = new DoubleLinkedList();
12
13          // Input values
14          int codeInput1 = -50;
15          int codeInput2 = 200;
16          int codeInput3 = 15;
17
18          // Expected outputs
19          int expectedOutput1 = -50; // peek after first insert
20          boolean expectedOutput2 = false;
21          int expectedOutput3 = 0; // peek after remove (empty)
21          boolean expectedOutput4 = true;
22
23          int expectedOutput5 = 200;
24          boolean expectedOutput6 = false;
25
26          int expectedOutput7 = 15;
27          boolean expectedOutput8 = false;
28
29          int expectedOutput9 = 200; // peek after removeFirst again
30          boolean expectedOutput10 = false;
31
32          try {
33              list.addFirst(codeInput1);
34              assertEquals(expectedOutput1, list.peekFirst());
35              assertEquals(expectedOutput2, list.isEmpty());
36
37              list.removeFirst();
38              assertEquals(expectedOutput3, list.isEmpty() ? 0 : list.peekFirst());
39              assertEquals(expectedOutput4, list.isEmpty());
40
41              list.addFirst(codeInput2);
42              assertEquals(expectedOutput5, list.peekFirst());
43              assertEquals(expectedOutput6, list.isEmpty());
44
45              list.addFirst(codeInput3);
46              assertEquals(expectedOutput7, list.peekFirst());
47              assertEquals(expectedOutput8, list.isEmpty());
48
49              list.removeFirst();
50              assertEquals(expectedOutput9, list.peekFirst());
51              assertEquals(expectedOutput10, list.isEmpty());
52
53          } catch (AssertionError ae) {
54              System.out.println(ae);
55          }
56      }
57
58      @Test
59      // addLast-removeLast-addFirst-addLast-removeLast
60      public void testDLL2() {
61          DoubleLinkedList list = new DoubleLinkedList();
62
63          // Input values
64          int codeInput1 = 120;
65          int codeInput2 = -30;
66          int codeInput3 = 75;
67
68          // Expected outputs
69          int expectedOutput1 = 120; // peek after addLast(120)
70          boolean expectedOutput2 = false; // isEmpty after addLast(120)
71
```

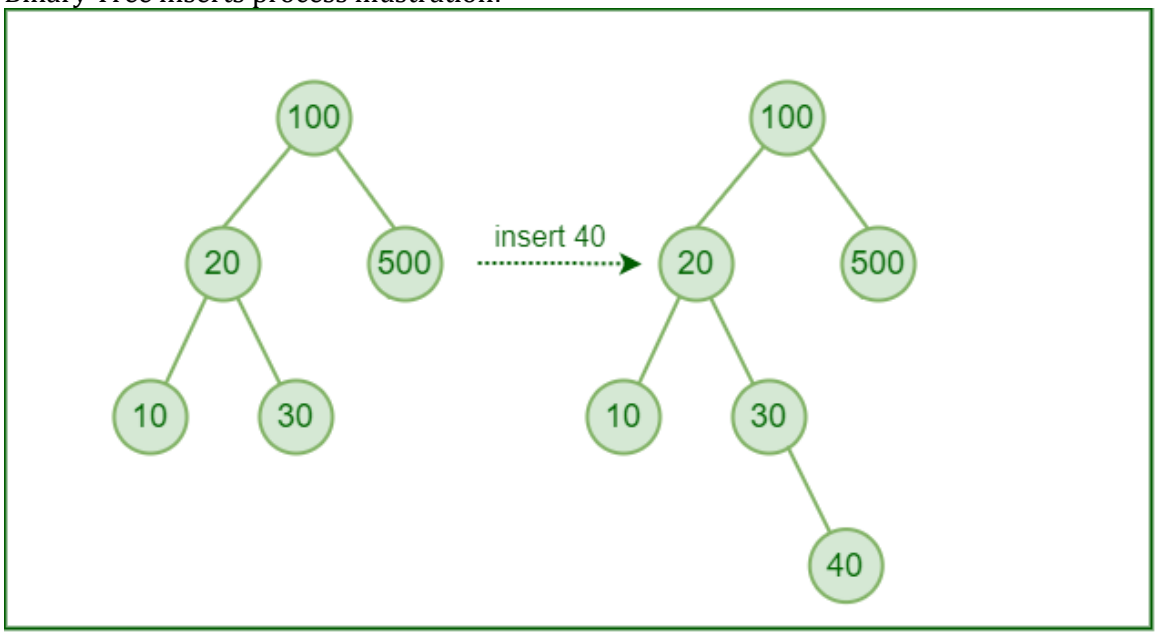
```
72          int expectedOutput3 = 0; // peek after removeLast (empty)
73          boolean expectedOutput4 = true; // isEmpty after removeLast
74
75          int expectedOutput5 = -30; // peek after addFirst(-30)
76          boolean expectedOutput6 = false; // isEmpty after addFirst(-30)
77
78          int expectedOutput7 = -30; // peek after addLast(75)
79          int expectedOutput8 = -30; // peek after removeLast
80
81          boolean expectedOutput9 = false; // isEmpty after removeLast
82
83          try {
84              // Step 1: addLast(120)
85              list.addLast(codeInput1);
86              assertEquals(expectedOutput1, list.peekFirst());
87              assertEquals(expectedOutput2, list.isEmpty());
88
89              // Step 2: removeLast()
90              list.removeLast();
91              assertEquals(expectedOutput3, list.isEmpty() ? 0 : list.peekFirst());
92              assertEquals(expectedOutput4, list.isEmpty());
93
94              // Step 3: addFirst(-30)
95              list.addFirst(codeInput2);
96              assertEquals(expectedOutput5, list.peekFirst());
97              assertEquals(expectedOutput6, list.isEmpty());
98
99              // Step 4: addLast(75)
100             list.addLast(codeInput3);
102             assertEquals(expectedOutput7, list.peekFirst());
103
104             // Step 5: removeLast()
105             list.removeLast();
106             assertEquals(expectedOutput8, list.peekFirst());
107             assertEquals(expectedOutput9, list.isEmpty());
108
109         } catch (AssertionError ae) {
110             System.out.println(ae);
111         }
112     }
113
114     @Test
115     // removeFirst, addFirst, addLast, addFirst, removeFirst, removeLast,
116     // removeFirst.
117     public void testDLL3() {
118         DoubleLinkedList list = new DoubleLinkedList();
119
120         // Input values
121         int codeInput1 = 0;
122         int codeInput2 = 999;
123         int codeInput3 = -100;
124
125         // Expected outputs
126         boolean expectedOutput1 = true; // isEmpty after removeFirst (should not crash)
127         int expectedOutput2 = 0; // peek after addFirst(0)
128         boolean expectedOutput3 = false; // isEmpty after addFirst(0)
129         int expectedOutput4 = 0; // peek after addLast(999)
130         boolean expectedOutput5 = false; // isEmpty after addLast(999)
131         int expectedOutput6 = -100; // peek after addFirst(-100)
132         boolean expectedOutput7 = false; // isEmpty after addFirst(-100)
133         int expectedOutput8 = 0; // peek after removeFirst()
134         boolean expectedOutput9 = false; // isEmpty after removeFirst()
135         int expectedOutput10 = 0; // peek after removeLast()
136         boolean expectedOutput11 = true; // isEmpty after removeLast()
137
138         try {
139             // Step 1: removeFirst() (should not crash)
140             list.removeFirst();
141             assertEquals(expectedOutput1, list.isEmpty());
142
143             // Step 2: addFirst(0)
```

```
144             list.addFirst(codeInput1);
145             assertEquals(expectedOutput2, list.peekFirst());
146             assertEquals(expectedOutput3, list.isEmpty());
147 
148             // Step 3: addLast(999)
149             list.addLast(codeInput2);
150             assertEquals(expectedOutput4, list.peekFirst());
151             assertEquals(expectedOutput5, list.isEmpty());
152 
153             // Step 4: addFirst(-100)
154             list.addFirst(codeInput3);
155             assertEquals(expectedOutput6, list.peekFirst());
156             assertEquals(expectedOutput7, list.isEmpty());
157 
158             // Step 5: removeFirst()
159             list.removeFirst();
160             assertEquals(expectedOutput8, list.peekFirst());
161             assertEquals(expectedOutput9, list.isEmpty());
162 
163             // Step 6: removeLast()
164             list.removeLast();
165             assertEquals(expectedOutput10, list.peekFirst());
166             assertEquals(expectedOutput11, list.isEmpty());
167 
168             // Step 7: removeFirst()
169             list.removeFirst();
170             assertEquals(expectedOutput11, list.isEmpty());
171 
172         } catch (AssertionError ae) {
173             System.out.println(ae);
174         }
175     }
176 
177 }
```

3.12 Binary Tree

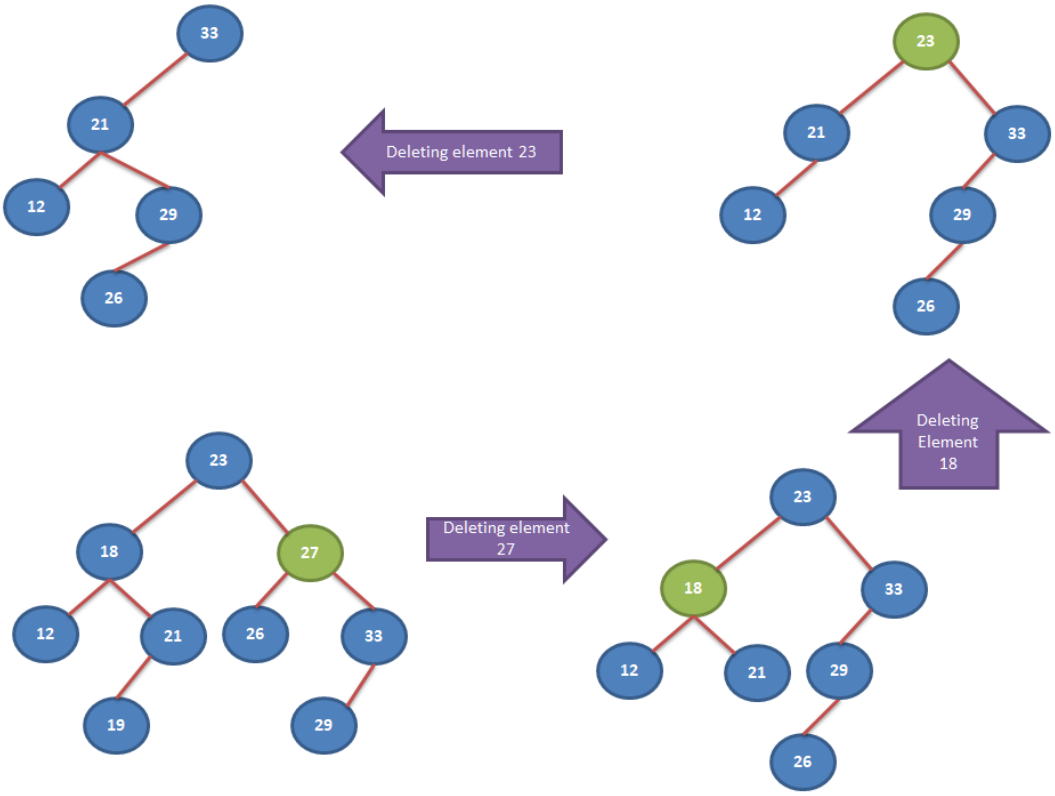
Please create source code based on the test code below.

Binary Tree inserts process illustration:



(Source : https://venkys.io/articles/details/binary-search-tree-insertion)

Binary Tree delete process illustraion:



(Source : https://simpletechtalks.com/binary-search-tree-deletion-of-node-explained-with-simple-example/)

Test code for Binary Tree class

BinaryTreeTest.java

```
 1  package dataStructure;
 2
 3  import static org.junit.Assert.*;
 4  import org.junit.Test;
 5
 6  public class BinaryTreeTest {
 7
 8      @Test
 9      public void testBST1() {
10          BinaryTree bst = new BinaryTree();
11
12          int codeInput1 = 50;
13          int codeInput2 = 30;
14          int codeInput3 = 70;
15          int codeInput4 = 20;
16          int codeInput5 = 40;
17          int codeInput6 = 60;
18          int codeInput7 = 80;
19
20          try {
21              // Insert elements
21              bst.insert(codeInput1);
22              bst.insert(codeInput2);
23              bst.insert(codeInput3);
24              bst.insert(codeInput4);
```

```
            bst.insert(codeInput5);
            bst.insert(codeInput6);
            bst.insert(codeInput7);

            // Searches
            assertTrue(bst.search(codeInput1));
            assertFalse(bst.search(100));
            assertTrue(bst.search(codeInput2));
            assertTrue(bst.search(codeInput3));

            // Delete and re-check
            bst.delete(codeInput2); // Remove 30
            assertFalse(bst.search(codeInput2));
            assertTrue(bst.search(codeInput5)); // 40 still exists

        } catch (AssertionError ae) {
            System.out.println(ae);
        }
    }

    @Test
    public void testBST2() {
        BinaryTree bst = new BinaryTree();

        int codeInput1 = 50;
        int codeInput2 = 30;
        int codeInput3 = 70;
        int codeInput4 = 20;
        int codeInput5 = 40;

        try {
            // Delete from empty
            bst.delete(50);
            assertTrue(bst.isEmpty());

            // Search in empty
            assertFalse(bst.search(50));
            assertFalse(bst.search(30));

            // Insert elements
            bst.insert(codeInput1);
            bst.insert(codeInput2);
            bst.insert(codeInput3);
            bst.insert(codeInput4);
            bst.insert(codeInput5);

            assertTrue(bst.search(codeInput5)); // 40
            assertTrue(bst.search(codeInput4)); // 20

            bst.delete(codeInput5); // Remove 40
            bst.delete(codeInput4); // Remove 20

            assertFalse(bst.search(codeInput5));
            assertFalse(bst.search(codeInput4));
            assertFalse(bst.isEmpty()); // Tree still has nodes

        } catch (AssertionError ae) {
            System.out.println(ae);
        }
    }

    @Test
    public void testBST3() {
        BinaryTree bst = new BinaryTree();

        int codeInput1 = 25;
        int codeInput2 = 15;
        int codeInput3 = 35;
        int codeInput4 = 20;

        try {
```

```
 96              bst.insert(codeInput1);
 97              assertTrue(bst.search(codeInput1));
 98
 99              assertFalse(bst.search(100));
100
102             bst.insert(codeInput2);
103             bst.insert(codeInput3);
104             assertTrue(bst.search(codeInput2));
105
106             bst.delete(codeInput2);
107             assertFalse(bst.search(codeInput2));
108
109             bst.insert(codeInput4);
110             assertTrue(bst.search(codeInput4));
111
112             bst.delete(codeInput3);
113             assertFalse(bst.search(codeInput3));
114
115         } catch (AssertionError ae) {
116             System.out.println(ae);
117         }
118     }
119 }
```

4. How to Run Java Tests in Visual Studio Code

1. Install Visual Studio Code from https://code.visualstudio.com

2. Install the Java Extension Pack from the Extensions Marketplace

3. Create a new folder for your Java project and open it in VS Code

4. Add your `.java` files inside the folder `dataStructure` (e.g., Calculator.java and CalculatorTest.java)

5. Use the integrated terminal:

   javac -cp .:junit-4.13.2.jar:hamcrest-core-1.3.jar *.java

   java -cp .:junit-4.13.2.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest

   or

   put .jar files inside the folder “lib” like the pictures below. You can directly download .jar

   files from this link https://github.com/mustmentari/test-cases-ASD-/tree/main

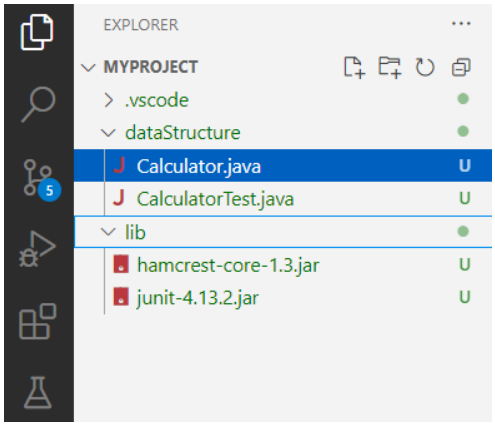
6. Create a java testing file in the package you have created. Copy and paste the test code provided in this guidebook.

7. Create another new java file to store your source. Then, create source code according to the test code in each question.

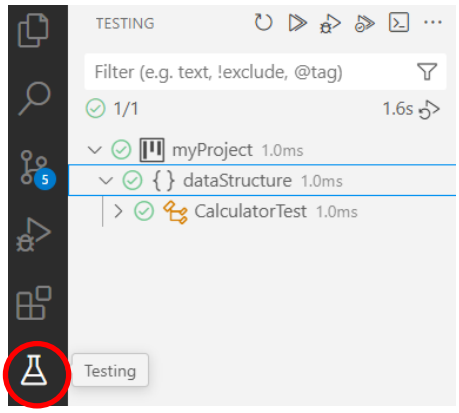
8. Analyze the output to debug and refine your implementation

📷 Example Screenshot:

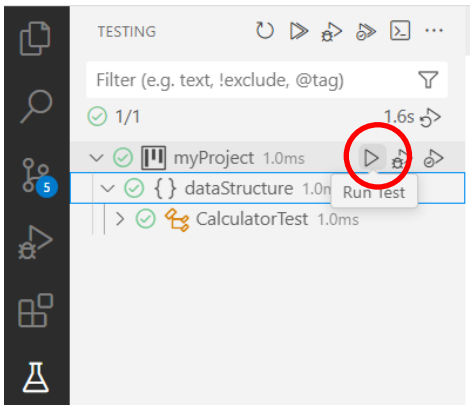
The image below shows the file structure within the project.



Then, click the testing icon according to the part circled in red in the image below. If this icon does not appear, make sure you have run the project.



Then, click run test according to the section circled in red in the image below to ensure that your source code matches the existing test code. If there is a green checklist mark shown according to the image, then your source code is correct, if a red cross appears, it means that the source code you created is not in accordance with the existing test code.



5. Assignment Submission Guidelines

Submission Deadline:

Deadline for problems 1-6

- **Date:** April 28, 2025
- **Time:** 11:59 PM (WIB)

Deadline for problems 7-12

- **Date:** Mei 8, 2025
- **Time:** 11:59 PM (WIB)

Submission Steps:

1. **Prepare Your Assignment File**
   Ensure your assignment file is in the correct format and is complete. The file name should follow this format:
   `Assignment1_FullName_Date`

   Example: `Assignment1_MustikaMentari_28Apr2025`

2. **Create a Google Drive Folder**
   Create a new folder in Google Drive with the following name:
   `Assignment [Course/Topic] - [Full Name]`

   Example: `Assignment Programming 1 - Mustika Mentari`

3. **Upload Your File**
   After creating the folder (Your project folder), upload your assignment file into that folder.
   Make sure the uploaded file is the correct one and follow the instructions given.
4. **Share the Folder**
   Once the file is uploaded, make sure that the folder/file can be accessed by the instructor or teaching assistants. To do so:
   - Right-click on the folder or file, select **"Get link"**.
   - Set the access to **"Anyone with the link"** and make sure the permission is set to **"Viewer"**.
5. **Send the Google Drive Link**
   Send the link to your Google Drive folder to google formulir on this link https://forms.gle/yvs2M1MbExb4a9B57.