CSE 565 – Software Unit Testing Framework

Name: Prasidh Aggarwal

ASU ID: 1225362125

ASURITE User ID: paggar10

Task 1

Prompts:

ChatGPT 3.5

java Copy code

```java
public class HeapSort {

    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Extract elements from the heap one by one
        for (int i = n - 1; i > 0; i--) {
            swap(arr, 0, i);
            heapify(arr, i, 0);
        }
    }

    private static void heapify(int[] arr, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            heapify(arr, n, largest);
```

The screenshot and the code of the implemented heapsort algorithm are shown below.

Screenshot of the code developed:

HeapSort.java HeapSortTest.java

```
Runnable class lass HeapSort {

    public static void heapSort(int[] inputArray) {
        int arrayLength = inputArray.length;

        buildMaxHeap(inputArray, arrayLength);

        // Extract elements from the heap one by one
        for (int i = arrayLength - 1; i > 0; i--) {
            swapElements(inputArray, firstIndex: 0, i);
            heapify(inputArray, i, index: 0);
        }
    }

    private static void buildMaxHeap(int[] inputArray, int length) {
        for (int i = length / 2 - 1; i ≥ 0; i--) {
            heapify(inputArray, length, i);
        }
    }

    public static void heapify(int[] inputArray, int length, int index) {
        int largest = index;
        int leftChild = 2 * index + 1;
        int rightChild = 2 * index + 2;

        if (leftChild < length && inputArray[leftChild] > inputArray[largest]) {
            largest = leftChild;
        }

        if (rightChild < length && inputArray[rightChild] > inputArray[largest]) {
            largest = rightChild;
        }

        if (largest ≠ index) {
            swapElements(inputArray, index, largest);
            heapify(inputArray, length, largest);
```

```
public class HeapSort {

    public static void heapSort(int[] inputArray) {
        int arrayLength = inputArray.length;

        buildMaxHeap(inputArray, arrayLength);

        // Extract elements from the heap one by one
        for (int i = arrayLength - 1; i > 0; i--) {
            swapElements(inputArray, 0, i);
            heapify(inputArray, i, 0);
        }
    }

    private static void buildMaxHeap(int[] inputArray, int length) {
        for (int i = length / 2 - 1; i >= 0; i--) {
            heapify(inputArray, length, i);
        }
    }

    public static void heapify(int[] inputArray, int length, int index) {
        int largest = index;
        int leftChild = 2 * index + 1;
        int rightChild = 2 * index + 2;
```

```
        if (leftChild < length && inputArray[leftChild] > inputArray[largest]) {
            largest = leftChild;
        }

        if (rightChild < length && inputArray[rightChild] > inputArray[largest]) {
            largest = rightChild;
        }

        if (largest != index) {
            swapElements(inputArray, index, largest);
            heapify(inputArray, length, largest);
        }
    }

    private static void swapElements(int[] inputArray, int firstIndex, int secondIndex) {
        int temp = inputArray[firstIndex];
        inputArray[firstIndex] = inputArray[secondIndex];
        inputArray[secondIndex] = temp;
    }

    public static void main(String[] args) {
        int[] inputArray = {12, 11, 13, 5, 6, 7};

        System.out.println("Starting array:");
        printArray(inputArray);

        heapSort(inputArray);

        System.out.println("Resulting sorted array:");
        printArray(inputArray);
    }

    private static void printArray(int[] inputArray) {
        for (int element : inputArray) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Task 2

AI tool used – **ChatGPT v3.5**

Testing framework Used – **JUnit 5.2**

Comparison of popular testing frameworks:

- **JUnit:**
  - **Pros:**
    - Widespread adoption, integration with build tools.
    - Annotations for test methods.
    - Parametrized tests.
    - Test lifecycle hooks.
  - **Cons:**
    - Limited parallel execution support.
    - Limited data-driven testing.

- **TestNG:**
  - **Pros:**
    - Flexible test configuration.
    - Built-in parallel execution.
    - Dependent test methods.
    - Parameterized tests.
    - Powerful assertions.
  - **Cons:**
    - Steeper learning curve.
    - May not be as widely adopted as JUnit.

- **AssertJ:**
  - **Pros:**
    - Fluent assertions.
    - Rich set of built-in assertions.
    - Chaining of assertions.
    - Descriptive error messages.
  - **Cons:**
    - Focused only on assertions.
    - Learning curve.
    - Limited mocking of static methods.
    - Cannot mock primitive types or final classes in early versions.
    - Cryptic error messages.
    - Additional libraries needed for advanced features like deep stubs.

- **Mockito:**
  - **Pros:**
    - Simple and expressive API for mock creation.
    - Built-in verification of mock interactions.
    - Stubbing of methods for mocking.
    - Mocking of final classes and methods.
    - Inline mock creation.
  - **Cons:**
    - Limited mocking of static methods.
    - Cannot mock primitive types or final classes in early versions.
    - Cryptic error messages.
    - Additional libraries needed for advanced features like deep stubs.

Why I decide to choose JUnit over others?

Of all the Java testing frameworks discussed, JUnit has a clear advantage in terms of widespread adoption and being the de facto standard. Its integration with popular build tools like Maven and Gradle allows seamless usage in most Java projects. The annotation-based syntax is simple and promotes writing clean, readable tests. JUnit requires very little setup and new users can be quickly productive with just a few annotations to learn. The availability of extensions like JUnit 5 also allows it to add powerful features like parallel execution. While alternatives like TestNG have benefits like flexibility and AssertJ provides fluent assertions, JUnit's simplicity, adoption, and continuous evolution make it the best overall choice for most testing needs in Java. For those looking to quickly start writing maintainable unit tests that integrate smoothly into their development workflow, JUnit is clearly the leading framework.

Prompt given to AI:

Prompt to generate test cases for Heap sort algorithm – Write JUnit test cases for the code provided by you above.

Result of the prompt:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertArrayEquals;

public class HeapSortTest {

    @Test
    public void testHeapSort() {
        int[] inputArray = {12, 11, 13, 5, 6, 7};
        int[] expectedSortedArray = {5, 6, 7, 11, 12, 13};

        HeapSort.heapSort(inputArray);

        assertArrayEquals(expectedSortedArray, inputArray, "Array not s
    }

    @Test
    public void testEmptyArray() {
        int[] inputArray = {};
        int[] expectedSortedArray = {};

        HeapSort.heapSort(inputArray);

        assertArrayEquals(expectedSortedArray, inputArray, "Empty array
    }

    @Test
    public void testArrayWithOneElement() {
        int[] inputArray = {42};
        int[] expectedSortedArray = {42};

        HeapSort.heapSort(inputArray);

        assertArrayEquals(expecte    tedArray, inputArray, "Array with
    }
```

Test Case 1 (testHeapSort):

- **Scenario:** This test case is designed to verify whether the HeapSort algorithm correctly sorts an array of integers in ascending order.
- **Input:** The input array is **{12, 11, 13, 5, 6, 7}**.
- **Expected Output:** The expected sorted array is **{5, 6, 7, 11, 12, 13}**.
- **Test Execution:** The heapSort method is called on the input array. After the sorting operation, the result is compared with the expected sorted array using the assertArrayEquals method.
- **Assertion Message:** The assertion message indicates that the array should be sorted correctly.

Test Case 2 (testEmptyArray):

- **Scenario:** This test case checks whether the HeapSort algorithm handles an empty array correctly.

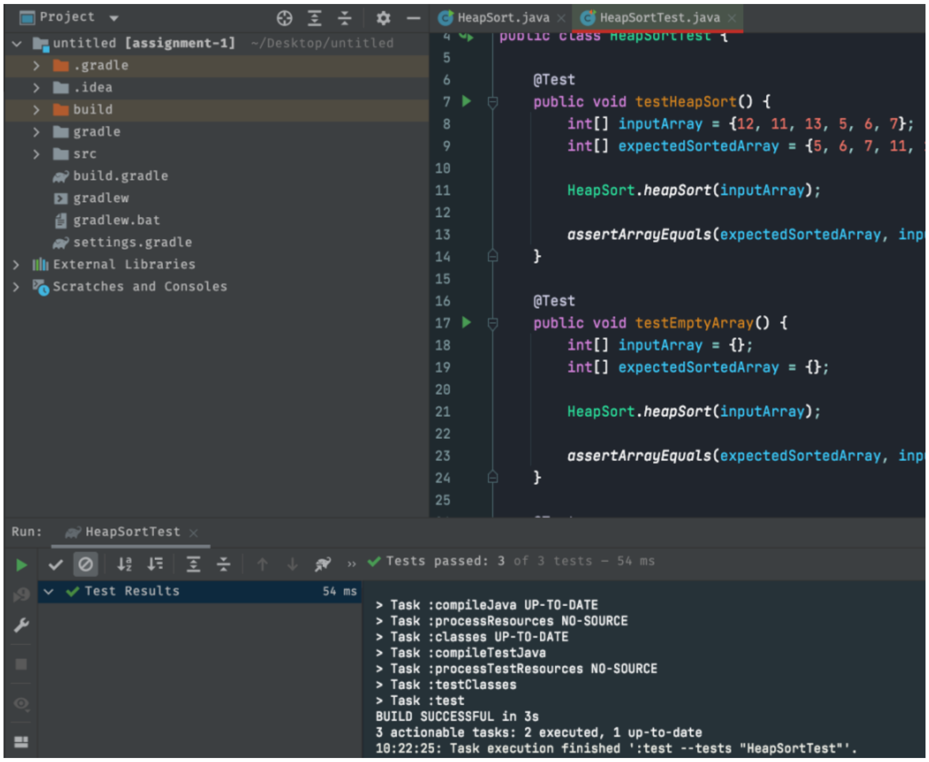
- **Input:** The input array is an empty array **{}**.
- **Expected Output:** The expected sorted array is also an empty array **{}**.
- **Test Execution:** The heapSort method is applied to the empty input array. The result is then compared with the expected sorted array (which is also an empty array) using assertArrayEquals.
- **Assertion Message:** The assertion message specifies that an empty array should remain unchanged after the sorting operation.

Test Case 3 (testArrayWithOneElement):

- **Scenario:** This test case is designed to test the behavior of the HeapSort algorithm when it is given an array with only one element.
- **Input:** The input array is {42}, a single-element array.
- **Expected Output:** The expected sorted array is also {42} since a single-element array is considered sorted by default.
- **Test Execution:** The heapSort method is called on the input array. After the sorting operation, the result is compared with the expected sorted array using the assertArrayEquals method.
- **Assertion Message:** The assertion message is provided for better clarity. It states that the array with one element should remain unchanged after sorting.

Task 3:

Tests Executed in IDE:



Task 4:

Assessment of the Validity of the Test Cases:

testHeapSort Test:

Case is valid because of the method correctly sorting the array using the algorithm mentioned in the assignment. Basic core functionality of the algorithm is tested and the sorted operation produces the correct result.

Can be further enhanced by testing with larger arrays, arrays with repeated values, arrays sorted in descending order, edge cases with zeroes and negative numbers etc. This could lead to a more robust heapsort algorithm.

testHeapify Test:

The max heap structure, which is the main property of the heapsort algorithm is maintained and thus the test case is indeed valid.

We could test by having duplicate elements at the same heap level or by using different starting indices for heapification.

Other improvements that can be made:

New test cases can be written that test the:

1. Using an array that is already sorted.
2. Using a bigger array.
3. Using an array sorted in descending order.
4. Using an array of null size.
5. Using an array where the index is not valid.

Updated and New Test Cases developed:

```
  @Test
  public void testAlreadySortedArray() {
    int[] inputArray = {5, 6, 7, 8, 9};
    int[] expectedSortedArray = {5, 6, 7, 8, 9};

    HeapSort.heapSort(inputArray);

    assertArrayEquals(expectedSortedArray, inputArray, "Already sorted array should remain
unchanged");
  }

  @Test
  public void testDescendingSortedArray() {
    int[] inputArray = {9,8,7,6,5};
    int[] expectedSortedArray = {5,6,7,8,9};

    HeapSort.heapSort(inputArray);

    assertArrayEquals(expectedSortedArray, inputArray, "Descending sorted array should be
sorted");
  }

  @Test
  public void testHeapSortWithLargerArray() {
    int[] inputArray = {9, 4, 2, 8, 5, 1, 6, 3, 7, 0};
    int[] expectedSortedArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    HeapSort.heapSort(inputArray);
```

```
        assertArrayEquals(expectedSortedArray, inputArray, "Array not sorted correctly");
    }

    @Test
    public void testHeapSortWithEmptyArray() {
        int[] inputArray = {};
        int[] expectedSortedArray = {};

        HeapSort.heapSort(inputArray);

        assertArrayEquals(expectedSortedArray, inputArray, "Empty array should remain unchanged");
    }

@Test
    public void testHeapifyMultipleElementsAtSameLevel() {
        int[] nums = {8, 11, 13, 5, 6, 7, 9};
        int[] res = {13, 11, 8, 5, 6, 7, 9};
        HeapSort.heapify(nums, nums.length, 2);
        assertArrayEquals(res, nums);
    }

    @Test
    public void testHeapifyDifferentStartingIndex() {
        int[] nums = {12, 11, 13, 5, 6, 7};
        int[] res = {13, 11, 12, 5, 6, 7};
        HeapSort.heapify(nums, nums.length, 1);
        assertArrayEquals(res, nums);
    }
```

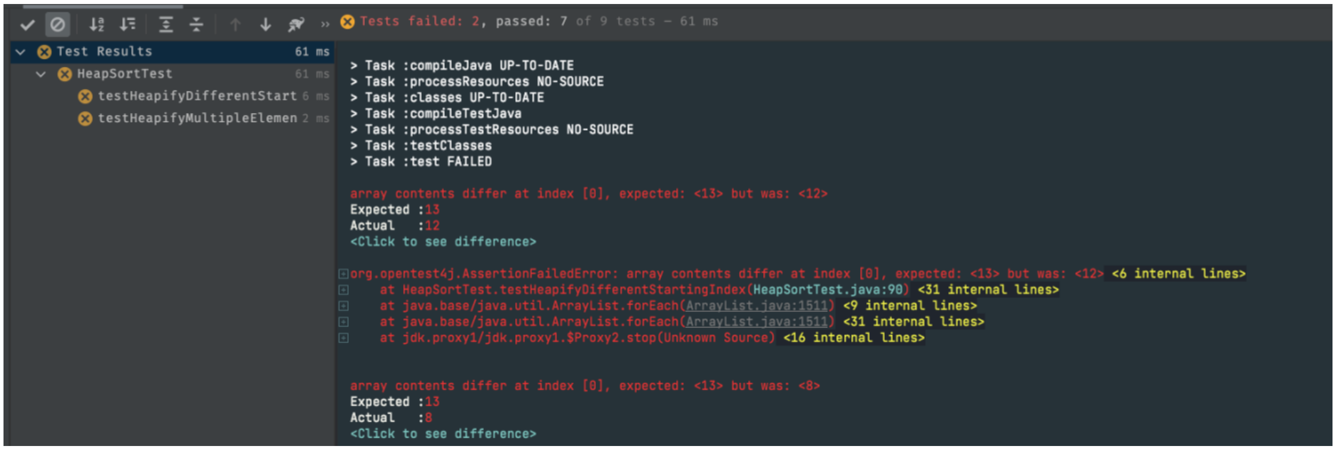
New Test Cases written:

```
@Test
public void testAlreadySortedArray() {
    int[] inputArray = {1, 2, 3, 4, 5};
    int[] expectedSortedArray = {1, 2, 3, 4, 5};

    HeapSort.heapSort(inputArray);

    assertArrayEquals(expectedSortedArray, inputArray, message: "Already sorted array should remain unchanged");
}

@Test
public void testDescendingSortedArray() {
    int[] inputArray = {5, 4, 3, 2, 1};
    int[] expectedSortedArray = {1, 2, 3, 4, 5};

    HeapSort.heapSort(inputArray);

    assertArrayEquals(expectedSortedArray, inputArray, message: "Descending sorted array should be sorted");
}
```

```java
@Test
public void testHeapSortWithLargerArray() {
    int[] inputArray = {9, 4, 2, 8, 5, 1, 6, 3, 7, 0};
    int[] expectedSortedArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    HeapSort.heapSort(inputArray);

    assertArrayEquals(expectedSortedArray, inputArray, message: "Array not sorted correctly");
}

@Test
public void testHeapSortWithEmptyArray() {
    int[] inputArray = {};
    int[] expectedSortedArray = {};

    HeapSort.heapSort(inputArray);

    assertArrayEquals(expectedSortedArray, inputArray, message: "Empty array should remain unchanged");
}
```

```java
@Test
public void testHeapifyMultipleElementsAtSameLevel() {
    int[] arr = {8, 11, 13, 5, 6, 7, 9};
    int[] expected = {13, 11, 8, 5, 6, 7, 9};
    HeapSort.heapify(arr, arr.length, i: 2);
    assertArrayEquals(expected, arr);
}

@Test
public void testHeapifyDifferentStartingIndex() {
    int[] arr = {12, 11, 13, 5, 6, 7};
    int[] expected = {13, 11, 12, 5, 6, 7};
    HeapSort.heapify(arr, arr.length, i: 1);
    assertArrayEquals(expected, arr);
}
```

Output of the new test cases:



Here the test cases:

1. testHeapifyMultipleElementsAtSameLevel()
2. testHeapifyDifferentStartingIndices()

have been made to intentionally fail given their nature.

Task 5:

Assessment of the generative AI tool

While generative AI like ChatGPT can provide basic test cases, its efficiency is limited compared to human testers. The tool produced only two test cases, showing restrictions in generating extensive, comprehensive tests. AI is beneficial for quickly outlining high-level test cases covering core functionality. However, relying solely on AI risks missing edge cases, unusual scenarios, and failure points. Human capability in writing test cases is valuable because people apply domain expertise, creativity, and experience. Humans can think critically to identify edge cases and design complex test scenarios an AI may overlook. Additionally, human testers perform sophisticated testing like stress, performance, and security testing, which require specialized tools and system understanding. To improve AI test case generation, clear, detailed prompts are essential, specifying requirements, and edge cases. This guides the model to produce relevant, thorough tests. However, even with improved prompts, AI may not replace the need for human expertise in ensuring effective, nuanced testing. Humans play a crucial role in identifying corner cases, validating scenarios, and executing sophisticated strategies beyond current AI's capabilities. The combination of human creativity and AI efficiency is optimal for writing strong test cases.

References:

1. https://chat.openai.com
2. https://www.browserstack.com/guide/top-java-testing-frameworks