

COIS 2020 Assignment 2 Part B Testing Document

Project: COIS 2020 — Assignment 2 Part B

Class under test: Point, and HashTable header

Testers: Martin Wang (0843637) and Henry Smith (0850896)

Date: November 10th, 2025

Step 1 - Complete Point class	
Test 1.1	Test Equals method
Description	Test that the Equals method returns True when 2 points equal each other
Input	<pre>Point p1 = new Point(1, 2); Point p2 = new Point(1, 5); // Same X, different Y Point p3 = new Point(2, 1); // Different X Point p4 = new Point(1, 2); // Identical to p1 // 1. Testing Equals Console.WriteLine(\$"p1 equals p4? {p1.Equals(p4)} (Expected: True)"); Console.WriteLine(\$"p1 equals p2? {p1.Equals(p2)} (Expected: False)");</pre>
Expected Output	p1 equals p4? True p1 equals p2? False
Actual Output	 <pre>--- Testing Point Class --- p1 equals p4? True (Expected: True) p1 equals p2? False (Expected: False)</pre>

Step 1 - Complete Point class	
Test 1.2	Test CompareTo method
Description	Test that when a point is smaller than the point it's being compared to, the CompareTo method returns a negative number. If it's bigger, then it will return a positive number. If they are equal, then 0 is returned.
Input	<pre>Console.WriteLine(\$"p1 compare p2: {p1.CompareTo(p2)} (Expected: Negative, as Y=2 < Y=5)"); Console.WriteLine(\$"p1 compare p3: {p1.CompareTo(p3)} (Expected: Negative, as X=1 < X=2)"); Console.WriteLine(\$"p3 compare p1: {p3.CompareTo(p1)} (Expected: Positive, as X=2 > X=1)"); Console.WriteLine(\$"p1 compare p4: {p1.CompareTo(p4)} (Expected: 0)");</pre>
Expected Output	-1 -1 1 0
Actual Output	 <pre>p1 compare p2: -1 (Expected: Negative, as Y=2 < Y=5) p1 compare p3: -1 (Expected: Negative, as X=1 < X=2) p3 compare p1: 1 (Expected: Positive, as X=2 > X=1) p1 compare p4: 0 (Expected: 0)</pre>

Step 1 - Complete Point class	
Test 1.3	Test GetHashCode method
Description	Test that get hash code works as expected and will return the same code for 2 points that are identical
Input	<pre>Console.WriteLine(\$"p1 GetHashCode: {p1.GetHashCode()}"); Console.WriteLine(\$"p4 GetHashCode: {p4.GetHashCode()}"); Console.WriteLine(\$"Hashes match? {p1.GetHashCode() == p4.GetHashCode()} (Expected: True)");</pre>

Step 1 - Complete Point class	
Expected Output	Equal hash code for both points
Actual Output	<p>✓</p> <pre>p1 HashCode: 9018 p4 HashCode: 9018 Hashes match? True (Expected: True)</pre>

Step 2 - Hash Table Header	
Test 1.1	Test HashTable and Node can be initiated without errors
Description	Test that the Hash Table successfully initiates without errors and that each bucket has a header node
Input	<p>Includes temporary test function:</p> <pre>public void DebugStructure() { int headerCount = 0; for (int i = 0; i < numBuckets; i++) { // Check if the bucket is NOT null (it // should have a header node) if (buckets[i] != null) { // Check if it's truly a header (Next // should be null initially) if (buckets[i].Next == null) { headerCount++; } } } if (headerCount == numBuckets) {</pre>

Step 2 - Hash Table Header

```

        Console.WriteLine($"SUCCESS: All
{numBuckets} buckets have a header node initialized.");
    }
    else
    {
        Console.WriteLine($"ERROR: Only
{headerCount} out of {numBuckets} buckets have header
nodes.");
    }
}

```

Main():

```

// 1. Instantiate the Table
    // Key = Point, Value = string (e.g., a label
for the point)
    int tableSize = 5;
    HashTable<Point, int> table = new
HashTable<Point, int>(tableSize);
    Console.WriteLine("HashTable instantiated.");

    // 2. Test Header Node Creation
    // This verifies the "InitializeBuckets" logic
    table.DebugStructure();

```

Expected Output

HashTable instantiates
Each bucket has a header node

Actual Output



```

--- Testing HashTable Structure ---
HashTable instantiated.
SUCCESS: All 5 buckets have a header node initialized.

```

Step 2 - Hash Table Header

Test 1.2

Test MakeEmpty method

Step 2 - Hash Table Header	
Description	Test that MakeEmpty does not crash the program. Since we use header nodes, empty means that every bucket should have exactly 1 header nodes pointing to nothing.
Input	<pre>Console.WriteLine("Calling MakeEmpty..."); table.MakeEmpty(); table.DebugStructure();</pre>
Expected Output	All 5 buckets have a header node initiated
Actual Output	<p>✓</p> <div style="border: 1px solid black; padding: 5px; background-color: #f0f0f0;"> Calling MakeEmpty... SUCCESS: All 5 buckets have a header node initialized. Test 1 </div>

Step 3 - Insert and Retrieve	
Test 1	Test that Insert and Retrieve methods behave as expected in sorted buckets under any sorted order and with collisions as well
Description	Make a table of size 1 to force collisions to happen, and insert the values in medium size, small size, then large size to ensure the sorted buckets behave as expected
Input	<pre>// TEST 1: SORTED INSERTION & COLLISIONS // We use a table size of 1 to FORCE collisions. // This proves your Linked List sorting logic works. Console.WriteLine("\n--- Test 1: Sorting & Collisions (Table Size 1) ---"); HashTable<Point, string> table = new HashTable<Point, string>(1); Point pMid = new Point(10, 10); Point pSmall = new Point(5, 5); Point pLarge = new Point(20, 20); try {</pre>

Step 3 - Insert and Retrieve

```
// Insert in mixed order: Mid, then Small,  
then Large  
    table.Insert(pMid, "Mid");           // List:  
[10]  
    table.Insert(pSmall, "Small");      // List: [5]  
-> [10]  
    table.Insert(pLarge, "Large");     // List: [5]  
-> [10] -> [20]  
  
    // Check if we can retrieve them  
    string val1 = table.Retrieve(pSmall);  
    string val2 = table.Retrieve(pMid);  
    string val3 = table.Retrieve(pLarge);  
  
    if (val1 == "Small" && val2 == "Mid" && val3  
== "Large")  
    {  
        Console.WriteLine("PASS: All items  
retrieved successfully despite collisions.");  
    }  
    else  
    {  
        Console.WriteLine("FAIL: Retrieved  
values did not match expected values.");  
    }  
}  
catch (Exception ex)  
{  
    Console.WriteLine($"FAIL: Exception during  
Test 1: {ex.Message}");  
}
```

Expected Output

PASS: All items retrieved successfully despite
collisions.

Actual Output

Step 3 - Insert and Retrieve	
	<pre>--- Test 1: Sorting & Collisions (Table Size 1) --- PASS: All items retrieved successfully despite collisions.</pre>

Step 4 - Duplicate Key Handling	
Test	Ensure that inserting a duplicate key fails
Description	Try to insert pMid again, which should fail. If it does, it's caught and a success message is printed
Input	<pre>Console.WriteLine("\n--- Test 2: Duplicate Key Handling ---"); try { table.Insert(pMid, "Duplicate Try"); // This should fail Console.WriteLine("FAIL: The table allowed a duplicate key insertion."); } catch (ArgumentException) { Console.WriteLine("PASS: Correctly blocked duplicate key with ArgumentException."); } catch (Exception ex) { Console.WriteLine(\$"FAIL: Wrong exception type thrown: {ex.GetType().Name}"); }</pre>
Expected Output	PASS: Correctly blocked duplicate key with ArgumentException.
Actual Output	<p>✓</p> <pre>--- Test 2: Duplicate Key Handling --- PASS: Correctly blocked duplicate key with ArgumentException.</pre>

Step 5 - Remove	
Test 1.1	Test the Remove function behaves as expected
Description	Remove pMid and then try to retrieve it to ensure that it doesn't exist anymore, then check that pSmall and pLarge still exist to ensure the chain is not broken after removal
Input	<pre>Console.WriteLine("\n--- Test 3: Remove Logic ---"); try { // Current State: [5] -> [10] -> [20] // 1. Remove the Middle Node (10, 10) table.Remove(pMid); // Verify Middle is gone try { table.Retrieve(pMid); Console.WriteLine("FAIL: Removed item (10,10) was still found!"); } catch { // This is good, we expect an error here } // Verify the chain is good (Small and Large should still exist) if (table.Retrieve(pSmall) == "Small" && table.Retrieve(pLarge) == "Large") { Console.WriteLine("PASS: Middle node removed, linked list remains intact."); } else { </pre>

Step 5 - Remove	
	<pre> Console.WriteLine("FAIL: Chain broke after removing middle node."); } } catch (Exception ex) { Console.WriteLine(\$"FAIL: Exception during remove test: {ex.Message}"); } </pre>
Expected Output	PASS: Middle node removed, linked list remains intact.
Actual Output	 <pre> --- Test 3: Remove Logic --- PASS: Middle node removed, linked list remains intact. </pre>

Step 6 - Missing Items	
Test	Properly handle retrieval and removal of points that do not exist in the table
Description	Try to retrieve and delete a point that does not exist in the hash table
Input	<pre> Console.WriteLine("\n--- Test 4: Retrieve/Remove Missing Items ---"); try { table.Retrieve(new Point(99, 99)); // Doesn't exist Console.WriteLine("FAIL: Retrieve(99,99) should have thrown an exception but didn't."); } catch (Exception) { Console.WriteLine("PASS: Retrieve correctly threw exception for missing item."); } </pre>

Step 6 - Missing Items	
	<pre> try { table.Remove(new Point(99, 99)); // Doesn't exist Console.WriteLine("FAIL: Remove(99,99) should have thrown an exception but didn't."); } catch (Exception) { Console.WriteLine("PASS: Remove correctly threw exception for missing item."); } Console.WriteLine("\nTests Complete."); } </pre>
Expected Output	PASS: Retrieve correctly threw exception for missing item. PASS: Remove correctly threw exception for missing item.
Actual Output	✓ <pre> --- Test 4: Retrieve/Remove Missing Items --- PASS: Retrieve correctly threw exception for missing item. PASS: Remove correctly threw exception for missing item. </pre>

Step 7 - Output	
Test	Test that output method correctly prints out items from smallest to biggest despite bucket order

Step 7 - Output	
Description	<p>First gather every node from every bucket in the hash table into a single, temporary list. Because the data is scattered across buckets based on hash codes, the method then sorts this list using a nested loop (Bubble Sort) that compares keys using the Point.CompareTo logic. Once the list is fully ordered from smallest to largest, the method iterates through it to print each key-value pair to the console.</p> <p>We create a new hash table and insert several points in a deliberately scrambled order (e.g., inserting a "Medium" point, then "Huge," then "Tiny"). The test then calls Output() to visually verify that the final printed list appears in the correct ascending order.</p>
Input	<pre>Console.WriteLine("\n--- Test 5: Output (Global Sort) ---"); // Create a fresh table HashTable<Point, string> outputTable = new HashTable<Point, string>(5); // Insert points in RANDOM order (not sorted) outputTable.Insert(new Point(10, 10), "Medium"); outputTable.Insert(new Point(100, 100), "Huge"); outputTable.Insert(new Point(1, 1), "Tiny"); outputTable.Insert(new Point(50, 50), "Large"); outputTable.Insert(new Point(5, 5), "Small"); Console.WriteLine("Calling Output()... (Visually check that order is Tiny -> Small -> Medium -> Large -> Huge)"); outputTable.Output();</pre>
Expected Output	The order that the points are printed should be smallest to greatest
Actual Output	✓

Step 7 - Output

```
--- Test 5: Output (Global Sort) ---
Calling Output()... (Visually check that order is Tiny -> Small -> Medium -> Large -> Huge)
--- Hash Table Contents (Sorted) ---
Key: (1, 1) | Value: Tiny
Key: (5, 5) | Value: Small
Key: (10, 10) | Value: Medium
Key: (50, 50) | Value: Large
Key: (100, 100) | Value: Huge
-----
```