public class **HeapSort**

{

decimal[] \_heapSortArray;

int \_heapSize;

public **HeapSort**(decimal[] heapSortArray)

{

if (heapSortArray.Length < 1) {

throw new ArgumentOutOfRangeException("The heap array must have … least one element.");

}

\_heapSortArray = heapSortArray;

\_heapSize = heapSortArray.Length;

}

public decimal[] HeapSortArray { get => \_heapSortArray; }

private int PARENT(int index) => Math.Abs(index / 2);

private int LEFT(int index) => index \* 2;

private int RIGHT(int index) => (index \* 2) + 1;

public void **MAX\_HEAPIFY**(int index)

{

if (\_heapSize < index) {

throw new ArgumentOutOfRangeException("The index value is less than … length.");

}

int left = LEFT(index);

int right = RIGHT(index);

int larger;

if (left < \_heapSize && GetNodeValue(left) > GetNodeValue(index)){

larger = left;

}

else {

larger = index;

}

if (right < \_heapSize && GetNodeValue(right) > GetNodeValue(larger)){

larger = right;

}

if (larger != index)

{

decimal auxLarger = \_heapSortArray[larger];

\_heapSortArray[larger] = \_heapSortArray[index];

\_heapSortArray[index] = auxLarger;

MAX\_HEAPIFY(larger);

}

}

private decimal GetNodeValue(int index)

{

try

{

return \_heapSortArray[index];

}

catch (IndexOutOfRangeException)

{

return 0;

}

catch(Exception err)

{

throw new Exception(err.Message);

}

}

public void **BUILD\_MAX\_HEAP**()

{

for (int i = Math.Abs(\_heapSize / 2); i > 0 ; i--)

{

**MAX\_HEAPIFY(i);**

}

}

public void **HEAPSORT**()

{

**BUILD\_MAX\_HEAP();**

int initialIndex = 1;

for (int i = \_heapSortArray.Length - 1; i > 0; i--)

{

decimal auxValue = \_heapSortArray[1];

\_heapSortArray[1] = \_heapSortArray[i];

\_heapSortArray[i] = auxValue;

initialIndex++;

\_heapSize -= 1;

**MAX\_HEAPIFY(1);**

}

}

}

**[TestMethod]**

public void HeapSortWithFifteenRepeatedElements()

{

// C# array is Zero-based because this its index initializes with number 0 and the arrange heapsort algorithm initializes with 1. One adaptation is necessary to match structure array with algorithm is initializes the heapSortArray in the position 1 and ignore the 0 position. Incresing heap array one position, so the zero position is ignored (-1).

decimal[] inHeapSortArray =

new decimal[16] { -1, 4, 1, 3, 2, 16, 9, 10, 14, 3, 7, 11, 12, 6, 5, 1 };

HeapSort heapSort = new **HeapSort**(inHeapSortArray);

heapSort.**HEAPSORT**();

Assert.AreEqual(heapSort.HeapSortArray[1], 1);

Assert.AreEqual(heapSort.HeapSortArray[2], 1);

Assert.AreEqual(heapSort.HeapSortArray[3], 2);

Assert.AreEqual(heapSort.HeapSortArray[4], 3);

Assert.AreEqual(heapSort.HeapSortArray[5], 3);

Assert.AreEqual(heapSort.HeapSortArray[6], 4);

Assert.AreEqual(heapSort.HeapSortArray[7], 5);

Assert.AreEqual(heapSort.HeapSortArray[8], 6);

Assert.AreEqual(heapSort.HeapSortArray[9], 7);

Assert.AreEqual(heapSort.HeapSortArray[10], 9);

Assert.AreEqual(heapSort.HeapSortArray[11], 10);

Assert.AreEqual(heapSort.HeapSortArray[12], 11);

Assert.AreEqual(heapSort.HeapSortArray[13], 12);

Assert.AreEqual(heapSort.HeapSortArray[14], 14);

Assert.AreEqual(heapSort.HeapSortArray[15], 16);

}