f) If you change the order of the numbers that are entered, do you get the same tree?

g) If the answer to part f is no, explain why it doesn't matter.

The order you enter the numbers generate a different tree depending on order.
This is because nodes that are less than their parent are entered from left to right. So entering nodes 4 and 3 under a parent node 5 will either be 3, 4 or 4, 3.

tutorial7 > BinaryMaxHeap.java > ...

```java
package tutorial7;

public class BinaryMaxHeap {

    protected Integer[] elements;

    // example
    /**
     * - - - - - - 16 - - - - - -
     * - - - - - / \ - - - - -
     * - - - - - 10 14 - - - - -
     * - - - - / \ / \ - - - -
     * - - - - 9 7 1 4 - - - -
     *
     * indices
     * - - - - - - 0 - - - - - -
     * - - - - - / \ - - - - -
     * - - - - - 1 2 - - - - -
     * - - - - / \ / \ - - - -
     * - - - - 3 4 5 6 - - - -
     *
     */

    public BinaryMaxHeap(int size) {
        this.elements = new Integer[size];
    }

    // parent (i)
    Integer parent(int index) {
        // returns the index of the parent
        return (int) Math.ceil((index - 1) / 2);
    }

    // leftChild(i)
    Integer leftChild(int index) {
        // returns the index of the left child
        if (index + 1 >= elements.length) {
            return null;
        }
        return index + 1;
    }

    // rightChild(i)
    Integer rightChild(int index) {
        // returns the index of the right child
        if (index + 2 >= elements.length) {
            return null;
        }
        return index + 2;
    }
```

```
51
52      // siftUp(i)
53      void siftUp(int index) {
54          // sifts up the element at index i
55          // until the heap property is restored
56          Integer parentIndex = parent(index);
57          if (parentIndex == null) {
58              // we're at the root
59              return;
60          }
61          Integer parentValue = elements[parentIndex];
62          Integer currentValue = elements[index];
63
64          if (parentValue < currentValue) {
65              // swap
66              elements[parent(index)] = currentValue;
67              elements[index] = parentValue;
68              siftUp(parent(index));
69          }
70      }
71
72      // insert(p)
73      void insert(int value) {
74          System.out.println("inserting " + value);
75          int index = findNextEmptyLeaf();
76          System.out.println("find next empty leaf: " + index);
77          elements[index] = value;
78
79          System.out.println("Starting siftup(" + index + ") of value=" + value);
80          siftUp(index);
81      }
82
83      // siftDown(i)
84      void siftDown(int index) {
85          System.out.println("Calling siftDown(" + index + ")");
86          // sifts down the element at index i
87          // until the heap property is restored
88          Integer leftChildIndex = leftChild(index);
89          Integer rightChildIndex = rightChild(index);
90
91          // 4 cases:
92          // both null,
93          // left null right non-null,
94          // left non-null right null
95          // both non-null
96
97          Integer currentValue = elements[index];
98          if (leftChildIndex == null && rightChildIndex == null) {
99              return;
100         } else if (leftChildIndex == null && rightChildIndex != null && elements
```

```
            // sift right if needed
            Integer rightChildValue = elements[rightChildIndex];
            if (currentValue < rightChildValue) {
                // swap
                elements[rightChildIndex] = currentValue;
                elements[index] = rightChildValue;
                siftDown(rightChildIndex);
            }
        } else if (leftChildIndex != null && rightChildIndex == null && elements
        [leftChildIndex] != null) {
            // sift left if needed
            Integer leftChildValue = elements[leftChildIndex];
            if (currentValue < leftChildValue) {
                // swap
                elements[leftChildIndex] = currentValue;
                elements[index] = leftChildValue;
                siftDown(leftChildIndex);
            }
        } else {
            // both non-null
            if (elements[leftChildIndex] != null && elements[rightChildIndex] != null)
            {
                Integer leftChildValue = elements[leftChildIndex];
                Integer rightChildValue = elements[rightChildIndex];
                if (leftChildValue > rightChildValue) {
                    // swap
                    elements[leftChildIndex] = currentValue;
                    elements[index] = leftChildValue;
                    siftDown(leftChildIndex);
                } else if (leftChildValue < rightChildValue) {
                    // swap
                    elements[rightChildIndex] = currentValue;
                    elements[index] = rightChildValue;
                    siftDown(rightChildIndex);
                }
            }
        }
    }

    int findNextEmptyLeaf() {
        int i = 0;
        while (i < elements.length) {
            if (elements[i] == null) {
                return i;
            }
            i++;
        }
        return i;
    }

```

```
149     // extractMax()
150     int extractMax() {
151         // returns the maximum element
152         // and removes it from the heap
153         Integer max = elements[0];
154         Integer leafIndex = findNextEmptyLeaf()-1;
155         elements[0] = elements[leafIndex];
156         elements[leafIndex] = null;
157         siftDown(index: 0);
158         return max;
159     }
160 
161     // remove(i)
162     void remove(int index) {
163         // removes the element at index i
164         elements[index] = 2000000000;
165         siftUp(index);
166         extractMax();
167     }
168 }
169 
```

Main.java

```java
package tutorial7;

public class Main {

    Run | Debug
    public static void main(String[] args) {
        int[] elements = { 16,
                10, 14,
                9, 7, 1, 4,
                2, 8, 3 };
        int nodes = calculateNodes(height: 4);

        BinaryMaxHeap bmh = new BinaryMaxHeap(nodes);
        for (int i : elements) {
            bmh.insert(i);
        }

        printBinaryMaxHeap(bmh);

        System.out.println(x: "\n\n");

        bmh.extractMax();

        printBinaryMaxHeap(bmh);

        bmh.remove(index: 1);

        System.out.println(x: "\n\n");

        printBinaryMaxHeap(bmh);

        System.out.println();

        System.out.println("Left child index of 0: " + bmh.leftChild(index: 0));
        System.out.println("Right child index of 0: " + bmh.rightChild(index: 0));

    }

    // calculates amount of space for a given tree height
    public static int calculateNodes(int height) {
        int sum = 0;
        for (int i = 0; i < height; i++) {
            sum += Math.pow(a: 2, i);
        }
        return sum;
    }

    static int PowerOf2(int power) {
        return (1 << power);
    }

}
```

```
51      public static void printBinaryMaxHeap(BinaryMaxHeap bmh) {
52          int currentLevel = 0;
53          int maxPerLevel = PowerOf2(currentLevel);
54          for (int i = 0; i < bmh.elements.length; i++) {
55              if (i == maxPerLevel - 1) {
56                  System.out.println(x: "\n");
57                  currentLevel++;
58                  maxPerLevel = PowerOf2(currentLevel);
59              }
60              System.out.print(" " + bmh.elements[i]);
61          }
62      }
63
64  }
65
```