Solution 1

```
  handler.rebalanceHeaps()
  handler.updateMedian()
}

func (handler *ContinuousMedianHandler) rebalanceHeaps() {
  if handler.lowers.Length()-handler.greaters.Length() == 2 {
    handler.greaters.Insert(handler.lowers.Remove())
  } else if handler.greaters.Length()-handler.lowers.Length() == 2 {
    handler.lowers.Insert(handler.greaters.Remove())
  }
}

func (handler *ContinuousMedianHandler) updateMedian() {
  if handler.lowers.Length() == handler.greaters.Length() {
    sum := (handler.lowers.Peek() + handler.greaters.Peek())
    handler.Median = float64(sum) / 2
  } else if handler.lowers.Length() > handler.greaters.Length() {
    handler.Median = float64(handler.lowers.Peek())
  } else {
    handler.Median = float64(handler.greaters.Peek())
  }
}

type Heap struct {
  comp   HeapFunc
  values []int
}

type HeapFunc func(int, int) bool

var MinHeapFunc = func(a, b int) bool { return a < b }
var MaxHeapFunc = func(a, b int) bool { return a > b }

func NewHeap(fn HeapFunc) *Heap {
  return &Heap{
    comp:   fn,
    values: []int{},
  }
}

func (h *Heap) Length() int {
  return len(h.values)
}

func (h *Heap) Peek() int {
  if len(h.values) == 0 {
    return -1
  }
  return h.values[0]
}

func (h *Heap) Insert(value int) {
  h.values = append(h.values, value)
  h.siftUp()
}

func (h *Heap) Remove() int {
  l := h.Length()
  h.swap(0, l-1)
  peeked := h.values[l-1]
  h.values = h.values[0 : l-1]
  h.siftDown()
  return peeked
}

func (h *Heap) siftUp() {
  currentIndex := h.Length() - 1
  parentIndex := (currentIndex - 1) / 2
  for currentIndex > 0 {
    current, parent := h.values[currentIndex], h.values[parentIndex]
    if h.comp(current, parent) {
      h.swap(currentIndex, parentIndex)
      currentIndex = parentIndex
      parentIndex = (currentIndex - 1) / 2
    } else {
      return
    }
  }
}

func (h *Heap) siftDown() {
  currentIndex := 0
  endIndex := h.Length() - 1
  childOneIdx := currentIndex*2 + 1
  for childOneIdx <= endIndex {
    childTwoIdx := -1
```