Solution 1

```
            var firstParentIndex = Double((array.count - 2) / 2)
            firstParentIndex = firstParentIndex.rounded(.down)

            for var currentIndex in (0 ... Int(firstParentIndex)).reversed() {
                var endIndex = array.count - 1
                siftDown(currentIndex: &currentIndex, endIndex: &endIndex, heap: &array)
            }

            return array
        }

        // O(log(n)) time | O(1) space
        func siftDown(currentIndex: inout Int, endIndex: inout Int, heap: inout [Int]) {
            var firstChildIndex = (2 * currentIndex) + 1
            while firstChildIndex < endIndex {
                var secondChildIndex = -1
                let potentialSecondChild = (2 * currentIndex) + 2
                if potentialSecondChild <= endIndex {
                    secondChildIndex = potentialSecondChild
                }

                var indexToSwap = -1
                if secondChildIndex != -1, heap[secondChildIndex] < heap[firstChildIndex] {
                    indexToSwap = secondChildIndex
                } else {
                    indexToSwap = firstChildIndex
                }

                if heap[indexToSwap] < heap[currentIndex] {
                    swap(firstIndex: currentIndex, secondIndex: indexToSwap, heap: &heap)
                    currentIndex = indexToSwap
                    firstChildIndex = (2 * currentIndex) + 1
                } else {
                    return
                }
            }
        }

        // O(log(n)) time | O(1) space
        func siftUp(currentIndex: inout Int, heap: inout [Int]) {
            var parentIndex = Double((currentIndex - 1) / 2)
            parentIndex = parentIndex.rounded(.down)

            while currentIndex > 0, heap[currentIndex] < heap[Int(parentIndex)] {
                swap(firstIndex: currentIndex, secondIndex: Int(parentIndex), heap: &heap)
                currentIndex = Int(parentIndex)
                parentIndex = Double((currentIndex - 1) / 2)
            }
        }

        // O(1) time | O(1) space
        func peek() -> Int {
            return heap[0]
        }

        // O(log(n)) time | O(1) space
        func remove() -> Int {
            swap(firstIndex: 0, secondIndex: heap.count - 1, heap: &heap)

            if let valueToRemove = heap.popLast() {
                var currentIndex = 0
                var endIndex = heap.count - 1
                siftDown(currentIndex: &currentIndex, endIndex: &endIndex, heap: &heap)

                return valueToRemove
            }

            return -1
        }

        // O(log(n)) time | O(1) space
        func insert(value: Int) {
            heap.append(value)
            var currentIndex = heap.count - 1
            siftUp(currentIndex: &currentIndex, heap: &heap)
        }

        // Generic swap function
        func swap(firstIndex: Int, secondIndex: Int, heap: inout [Int]) {
            let temp = heap[firstIndex]
            heap[firstIndex] = heap[secondIndex]
            heap[secondIndex] = temp
        }
    }
}
```