```lua
--A heap that sorts keys in a {key, value} pair set of data
-- such that keys are type(number)

local Heap = {}
Heap.mt = {} --metatable
Heap.prototype = {}
Heap.mt.__index = Heap.prototype

-- Utility Function
table.exchange = function(t, a, b)
    local tmp = t[a]
    t[a] = t[b]
    t[b] = tmp
end

--data := { { k = type(number), v = (Anything) }, ... }
function Heap.new(isMax,data)
    data = data or {}
    local heap = {}
    setmetatable(heap, Heap.mt)
    heap.isMax = isMax and true
    heap.heapsize = #data
    for i = 1, #data do
        heap[i] = data[i]
    end
    return heap
end

function Heap.prototype.parent(self,i)
    return math.max(1,bit32 and bit32.rshift(i,1) or math.floor(i/2))
end

function Heap.prototype.left(self,i)
    return math.max(1,bit32 and bit32.lshift(i,1) or 2*i)
end

function Heap.prototype.right(self,i)
    return math.max(1,bit32 and (bit32.lshift(i,1)+1) or (2*i+1))
end

function Heap.prototype.minHeapify(self,i)
    if self.isMax == true then error("Can't call minHeapify on max heap") end
    local l = self:left(i)
    local r = self:right(i)
    local smallest = nil
```

```lua
    if l <= self.heapsize and self[l].k < self[i].k then
        smallest = l
    else
        smallest = i
    end
    if r <= self.heapsize and self[r].k < self[smallest].k then
        smallest = r
    end
    if smallest ~= i then
        table.exchange(self,i,smallest)
        self.minHeapify(self,smallest)
    end
end

function Heap.prototype.buildMinHeap(self)
    self.isMax = false
    self.heapsize = #self
    for i = math.floor(#self), 1, -1 do
        self.minHeapify(self,i)
    end
end

function Heap.prototype.maxHeapify(self,i)
    if self.isMax ~= true then error("Can't call maxHeapify on min heap") end
    local l = self:left(i)
    local r = self:right(i)
    local largest = nil
    if l <= self.heapsize and self[l].k > self[i].k then
        largest = l
    else
        largest = i
    end
    if r <= self.heapsize and self[r].k > self[largest].k then
        largest = r
    end
    if largest ~= i then
        table.exchange(self,i,largest)
        self.maxHeapify(self,largest)
    end
end

function Heap.prototype.buildMaxHeap(self)
    self.isMax = true
    self.heapsize = #self
    for i = math.floor(#self), 1, -1 do
```

```lua
        self.maxHeapify(self,i)
    end
end

function Heap.prototype.heapsort(self)
    local heapify = nil
    if self.isMax == true then
        heapify = self.maxHeapify
        self.buildMaxHeap(self)
    else
        heapify = self.minHeapify
        self.buildMinHeap(self)
    end
    for i = #self, 2, -1 do
        table.exchange(self,1,i)
        self.heapsize = self.heapsize-1
        heapify(self,1)
    end
end

function Heap.prototype.size(self)
    return self.heapsize
end

---------------------------------------------------
-- Priority Queue methods
---------------------------------------------------
-- Max priority queue methods
function Heap.prototype.maximum(self,i)
    if self.isMax ~= true then
        --no maximum garanteed when using a min priority queue
        error("maximum(): invalid operation on min heap.")
    end
    return self[1]
end

function Heap.prototype.extractMax(self)
    if self.isMax ~= true then
        error("extractMax(): invalid operation on min heap.")
    end
    if self.heapsize < 1 then error("extractMax(): heap underflow") end
    local max = self[1]
    self[1] = self[self.heapsize]
    self.heapsize = self.heapsize - 1
    self:maxHeapify(1)
```

```lua
    return max
end

function Heap.prototype.increaseKey(self,i,key)
    if self.isMax ~= true then
        error("increaseKey(): invalid operation on min heap.")
    end
    if key < self[i].k then
        error "new key is smaller than current key"
    end
    self[i].k = key
    while i > 1 and self[self:parent(i)].k < self[i].k do
        table.exchange(self, i, self:parent(i))
        i = self:parent(i)
    end
end

--
-- Min priority queue methods
--
function Heap.prototype.minimum(self)
    if self.isMax == true then
        --no maximum garanteed when using a min priority queue
        error("minimum(): invalid operation on max heap.")
    end
    return self[1]
end

function Heap.prototype.extractMin(self,i)
    if self.isMax == true then
        error("extractMin(): invalid operation on max heap.")
    end
    if self.heapsize < 1 then error("extractMin(): heap underflow") end
    local min = self[1]
    self[1] = self[self.heapsize]
    self.heapsize = self.heapsize - 1
    self:minHeapify(1)
    return min
end

function Heap.prototype.decreaseKey(self,i,key)
    if self.isMax == true then
        error("decreaseKey(): invalid operation on max heap.")
    end
    if key > self[i].k then
```

```lua
            error "new key is bigger than current key"
        end
        self[i].k = key
        while i > 1 and self[self:parent(i)].k > self[i].k do
            table.exchange(self, i, self:parent(i))
            i = self:parent(i)
        end
    end

    --
    -- Standard priority queue methods
    --
    function Heap.prototype.insert(self,key,value)
        self.heapsize = self.heapsize+1
        if self.isMax then
            self[self.heapsize] = {k = -math.huge, v = value}
            self:increaseKey(self.heapsize,key)
        else
            self[self.heapsize] = {k = math.huge, v = value}
            self:decreaseKey(self.heapsize,key)
        end
    end

    function Heap.prototype.removeKey(self,key)
        return self:remove("k",key)
    end

    function Heap.prototype.removeValue(self,value)
        return self:remove("v",value)
    end

    --Removes first node found with given key/value
    function Heap.prototype.remove(self, kOrV, obj)
        assert(kOrV=="k" or kOrV=="v", [[Heap:remove() generalizes pairs by 'k' or 'v',
     therefore you must use one of these as the first parameter of find().]])
        local index, pair = self:find(kOrV,obj)
        if index then
            self[index] = self[self.heapsize]
            self.heapsize = self.heapsize - 1
            local heapify = self.isMax and self.maxHeapify or self.minHeapify
            heapify(self,index)
            return pair
        end
        return nil
    end
```

```lua
    --Updates the first found [k,v] with the new key
    function Heap.prototype.updateKeyByValue(self,value,newKey)
        self:removeValue(value)
        self:insert(newKey,value)
    end

    function Heap.prototype.findKey(self,key)
        return self:find("k",key)
    end

    function Heap.prototype.findValue(self,value)
        return self:find("v",value)
    end

    --Returns the index and pair of the first k or v object you're looking for
    function Heap.prototype.find(self,kOrV,obj)
        assert(kOrV=="k" or kOrV=="v", [[Heap:find() generalizes pairs by 'k' or 'v',
     therefore you must use one of these as the first parameter of find().]])
        for i=1, self.heapsize do
            if self[i][kOrV] == obj then
                return i, self[i]
            end
        end
        return nil, nil
    end

    -- Debug Utility methods
    function Heap.prototype.print(self,m)
        local out = (m and (m..': ') or '') .. ('['..(self[1].k or '')..', '..
    (self[1].v or '') ..']')
        for i = 2, self.heapsize do
            out = out .. ', ' .. '['.. self[i].k .. ', ' .. self[1].v .. ']'
        end
        out = out .. ' {'
        for i = self.heapsize+1, #self do
            out = out .. ', ' .. '['.. self[i].k .. ', ' .. self[1].v .. ']'
        end
        out = out .. ' }'
        print(out)
    end

    return Heap
```