A heap is a Complete Binary Tree: Each level contains the maximum number of nodes, except possibly the last layer, which must be filled from left to right.

A diagram of a tree

AI-generated content may be incorrect.

Every node satisfies the heap property: For any given node C, if P is a parent node of C, then:

* For a max heap: the key of P should be greater than or equal to the key of C.
* For a min heap: the key of P should be less than or equal to the key of C.A diagram of a diagram

  AI-generated content may be incorrect.

A heap of n nodes has only 1 possible shape. So we can represent the heap using an array, as shown;

A diagram of a square and a square

AI-generated content may be incorrect.

Note the relationship between parent and children nodes:

1. If a node is placed at index i in array, then if the resultant index lies within length of the array:
   * It's left child would be at (2i+1)th position
   * Right one would be at (2i+2)the position
2. If a node is placed at index i in array, it's parent node would be located at floor((i-1)/2)th index.

A diagram of a child's relationship

AI-generated content may be incorrect.

**Preparing the interface:** There are three things we want to achieve with our heap data structure:

*Note: Here, we talk about min-heap. We'll later see that it can be easily extended to max-heap as well.*

1. Get the max or min key from the heap (depending whether it's min or max heap)
2. Add a new key into the heap
3. Remove the max or min key from the heap (depending whether it's min heap or max heap)

For the min (max) heap, the first item would be the min (max) key. So we're left with two operations:

function heappush(heap, newKey){} // adds newKey into min-heap named "heap"

function heappop(heap){} // removes smallest key from min-heap named "heap"

**Implementing heappush()** : To add a new key, first push the new key into the array. Compare the pushed item with it's parent. If parent is larger than the pushed item then swap. Continue swapping until a legal parent is found or we've reached top of the heap. Here's a visual guide for better reference:

A diagram of steps and steps

AI-generated content may be incorrect.

Here's the final implementation:

function heappush(heap, newKey){

heap.push(newKey); // push the new key

let curr = heap.length-1; // get the current index of pushed key

// keep comparing until we terminate or root reached

while(curr > 0){

let parent = Math.floor((curr-1)/2)

if( heap[curr] < heap[parent] ){

// quick swap

[ heap[curr], heap[parent] ] = [ heap[parent], heap[curr] ]

curr = parent// update the index of newKey

} else break// if no swap, break, since we heap is stable now

}

}

**Implementing heappop():** Use heappop() to remove the topmost item of the heap. So remove the first array item, but then which node should become root? Follow these steps (for a min-heap):

1. Swap the root node with last node (first item with last item in the array)
2. Remove the root node by popping the last item out of the array
3. Compare the new root node's key with it's children:
   * If key is less than both of it's children keys then heap is stable
   * Else, swap the key with the smaller child key
4. Repeat step 3 until last child is reached or the heap property is established.

We're following a similar process as heappush(), except we're trying to establish the heap-property in top to bottom fashion i.e. start with root and keep going till last child. In heappush() we followed opposite order i.e. start from last child and keep going till the root.

function heappop(heap){

const n = heap.length; // swap root with last node

[heap[0], heap[n-1]] = [ heap[n-1], heap[0]]

const removedKey = heap.pop();// remove the root i.e. the last item (because of swap)

let curr = 0;

// keep going till atleast left child is possible for current node

while(2\*curr + 1 < heap.length){

const leftIndex = 2\*curr+1;

const rightIndex = 2\*curr+2;

const minChildIndex = (rightIndex<heap.length && heap[rightIndex]<heap[leftIndex]) ? rightIndex :leftIndex;

if(heap[minChildIndex] < heap[curr]){

// quick swap, if smaller of two children is smaller than parent (min-heap)

[heap[minChildIndex], heap[curr]] = [heap[curr], heap[minChildIndex]]

curr = minChildIndex

} else break

}

return removedKey; // finally return the removed key

}

**Create a heap using an existing array**: Create an empty heap and do heappush() on each array item;

function heapify(arr){

const heap = []

for(let item of arr) heappush(heap, item)

return heap;

}

We can avoid using extra space for the new heap altogether. Why not just re-arrange the items of the array itself so that it satisfies the heap property? To do this we can follow a similar logic as we did for heap pop. We can look at the first node and compare to it's children to see if it's the smallest one, if not swap it with the smaller child. In fact let's create a function for that called percolateDown(), since we're moving downwards:

// follows same logic as heappush, except minor modifications

function percolateDown(heap, index){

let curr = index;

// keep going down till heap property is established

while(2\*curr + 1 < heap.length){

const leftIndex = 2\*curr+1;

const rightIndex = 2\*curr+2;

const minChildIndex = (rightIndex < heap.length && heap[rightIndex] < heap[leftIndex] ) ? rightIndex :leftIndex;

if(heap[minChildIndex] < heap[curr]){

// quick swap, if smaller of two children is smaller than parent (min-heap)

[heap[minChildIndex], heap[curr]] = [heap[curr], heap[minChildIndex]]

curr = minChildIndex

} else break

}

We can use percolateDown() for all array items to put everything in correct order as per heap-property:

function heapify(heap){

for(let i in heap) percolateDown(heap, i)

return heap

}

That saves us an extra array. But we're doing some repetitive work here. If there are n nodes in heap, out of which x are leaf nodes then we only need to perform percolateDown() for n-x nodes, since last x nodes would be in correct place by then.

So in the array representation of heap, till which index we should perform the percolateDown() operation? Well, till the index where parent of the last node lies. Because as soon as parent of last node is percolated down it'll take care of the last node too. So:

* If array length is n
* Last node's index would be: n-1
* It's parent node's index would be:

Math.floor((n-1) - 1 / 2) = Math.floor(n/2 - 1)

Hence our final heapify function would be:

function heapify(heap){

const last = Math.floor(heap.length/2 - 1);

for(let i = 0; i <= last; i++) percolateDown(heap, i)

return heap

}

**What about max-heap?** If we've an implementation of minHeap we can easily use it as a max heap as well. We just need to ensure that while adding values to the heap we insert negative of the key. It would ensure that heap acts as min-heap for negative of all the keys which is equivalent to maxHeap for all the actual keys. Example:

* Say we have an array const x = [23, 454, 54, 29];
* Min-heap can be created using:

const heap = [];

for(let el of x) heappush(heap, el);

const min = heappop(heap) // min value

* Max-heap can be created using:

const heap = [];

for(let el of x) heappush(heap, -el);

const max = -heappop(heap) // max value

===========================

class MinHeap {

constructor() { this.heap = []; }

getLeftChildIndex(parentIndex) { return 2 \* parentIndex + 1; }

getRightChildIndex(parentIndex) { return 2 \* parentIndex + 2; }

getParentIndex(childIndex) { return Math.floor((childIndex - 1) / 2); }

hasParent(index) { return this.getParentIndex(index) >= 0; }

swap(index1, index2) {

[this.heap[index1], this.heap[index2]] = [this.heap[index2],this.heap[index1]];

}

insert(value) {

this.heap.push(value);

this.heapifyUp();

}

heapifyUp() {

let currentIndex = this.heap.length - 1;

while (

this.hasParent(currentIndex) &&

this.heap[currentIndex] < this.heap[this.getParentIndex(currentIndex)]

) {

this.swap(currentIndex, this.getParentIndex(currentIndex));

currentIndex = this.getParentIndex(currentIndex);

}

}

removeMin() {

if (this.heap.length === 0) throw new Error("Heap is empty");

const minValue = this.heap[0];

this.heap[0] = this.heap.pop();

this.heapifyDown();

return minValue;

}

heapifyDown() {

let currentIndex = 0;

while (this.getLeftChildIndex(currentIndex) < this.heap.length) {

let smallerChildIndex = this.getLeftChildIndex(currentIndex);

if (this.getRightChildIndex(currentIndex)<this.heap.length &&

this.heap[this.getRightChildIndex(currentIndex)]<this.heap[smallerChildIndex]) {

smallerChildIndex = this.getRightChildIndex(currentIndex);

}

if (this.heap[currentIndex] < this.heap[smallerChildIndex]) break;

else this.swap(currentIndex, smallerChildIndex);

currentIndex = smallerChildIndex;

}

}

}

const priorityQueue = new MinHeap();

priorityQueue.insert(10);

priorityQueue.insert(5);

priorityQueue.insert(20);

priorityQueue.insert(3);

console.log("Removed min value:", priorityQueue.removeMin());

console.log("Removed min value:", priorityQueue.removeMin());