A priority queue is a special type of [queue](https://www.studytonight.com/data-structures/queue-data-structure), where the elements within the queue have a specified priority-level.

In a traditional queue, there is a First-In-First-Out (FIFO) rule, similar to queuing up in a shop. A priority queue would enable someone with a higher priority to jump the queue all and go all the way to the front of the queue.

### Contents

1. [Heaps](#heaps)

2. [Implementation](#implementation)

3. [Analysis](#analysis)

4. [Use Cases](#use-cases)

5. [Challenge](#challenge)

### Heaps

Heaps are a tree-based data structure, usually implemented as an array, which represent a priority queue. There are two types of heaps:

1. Min Heap

2. Max Heap

Once you know one of these, it is simple to know the other. In this article we will be looking at max-heaps. Afterwards, it would be a good exercise to see if you could implement a min-heap using the knowledge you have gained from this article.

![Alt Text](https://dev-to-uploads.s3.amazonaws.com/i/f39j4qzk4qtaisfo1rzz.png)

In the image above, you will see we have an array with the elements: 26, 24, 20, 18 & 17. This can be represented as the binary tree you also see in the picture above. Notice how each element is added to the tree from left to right. If a new element was to be inserted, it would become the left child of the node which has a priority level of 20, assuming the element had a lesser priority than this.

The first node, 26, is called the root node, in a max-heap this is the largest number, i.e. the node with the highest priority which should be extracted next. Each node can have a left child, and a right child. Notice how the children of each node are lesser than the priority value of the parent node. Each node, other than the root node, has a parent node, which is just one element up from the node you are looking at.

Elements with any priority value can be inserted in to the heap. With every insertion, there is an ordering completed to correctly position the newly inserted element.

Elements are \*dequeued/extracted\* from the root, which similar to insertion, is also followed by an ordering operation, in order to correctly position the next element with the highest priority at the root of the tree.

### Implementation

\*\*Public Methods:\*\*

\* insert

\* extractMax / extractMin

\* heapify

\* peek

\* swap

\*\*Helper Functions\*\*

``` javascript

const leftChild = (index) => index \* 2 + 1;

const rightChild = (index) => index \* 2 + 2;

const parent = (index) => Math.floor((index - 1) / 2);

```

\* In order to get the position of a left child of a node, we multiply the index by 2, and add 1.

\* In order to get the right child of a node, we multiply the index by 2, and add 2.

\* In order to get the parent of a node, we subtract our index by 1, and divide by 2. We round down any floating numbers we get from dividing an odd number.

\*\*Constructor\*\*

This initialises our heap as an empty array.

``` javascript

function maxHeap() {

this.heap = [];

}

```

\*\*Swap\*\*

This swaps swaps two elements in an array.

``` javascript

MaxHeap.prototype.swap = function (indexOne, indexTwo) {

const tmp = this.heap[indexOne];

this.heap[indexOne] = this.heap[indexTwo];

this.heap[indexTwo] = tmp;

}

```

\*\*Peek\*\*

This shows you the current root of the heap. It does not extract the root node from the tree.

``` javascript

maxHeap.prototype.peek = function() {

// the root is always the highest priority item

return this.heap[0];

}

```

\*\*Insert\*\*

This pushes an element on to our heap. After we have inserted the element, we correctly position the element by comparing the values of the newly inserted element with its parent. If the newly inserted element is greater, that is it has a greater priority, then the newly inserted element is swapped with its parent. This is recursively called until the element is in its right place.

``` javascript

maxHeap.prototype.insert = function(element) {

// push element to the end of the heap

this.heap.push(element);

// the index of the element we have just pushed

let index = this.heap.length - 1;

// if the element is greater than its parent:

// swap element with its parent

while (index !== 0 && this.heap[index] > this.heap[parent(index)]) {

this.swap(index, parent(index));

index = parent(index);

}

}

```

\*\*ExtractMax\*\*

This extracts the root from the heap and calls heapify to reposition the rest of the heap.

``` javascript

maxHeap.prototype.extractMax = function() {

// remove the first element from the heap

const root = this.heap.shift();

// put the last element to the front of the heap

// and remove the last element from the heap as it now

// sits at the front of the heap

this.heap.unshift(this.heap[this.heap.length - 1]);

this.heap.pop();

// correctly re-position heap

this.heapify(0);

return root;

}

```

\*\*Heapify\*\*

This re-positions the heap by comparing the left and right child of a specific node, and swapping them as necessary. This is recursively called until the heap is correctly re-positioned.

``` javascript

maxHeap.prototype.heapify = function(index) {

let left = leftChild(index);

let right = rightChild(index);

let smallest = index;

// if the left child is bigger than the node we are looking at

if (left < this.heap.length && this.heap[smallest] < this.heap[left]) {

smallest = left;

}

// if the right child is bigger than the node we are looking at

if (right < this.heap.length && this.heap[smallest] < this.heap[right]) {

smallest = right;

}

// if the value of smallest has changed, then some swapping needs to be done

// and this method needs to be called again with the swapped element

if (smallest != index) {

this.swap(smallest, index);

this.heapify(smallest);

}

}

```

### Use Cases

### Analysis

\* insert - O(log n)

\* peek - O(1)

\* extractMax / extractMin - O(log n)

### Challenge

Now see if you can create a min-heap. The methods are the same except the \*extractMax()\* method will be called \*extractMin()\*. The min-heap is the reverse of the max-heap, that is the smallest element is at the root of the heap.