**JavaScript堆排序算法**

1. 堆排序算法

堆排序（Heap sort）算法，是将数据看成近似完全二叉树结构，并根据完全二叉树的特性来进行排序的一种算法。完全二叉树要求每个结点的值都大于等于其左右子结点的值，称为大顶堆；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆。堆排序就是把先将父节点的最大数取出，并构建最大堆，再将堆继续调整为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。堆排序的平均时间复杂度为 O(n log2 n)，空间复杂度：O(1)，稳定性：不稳定。

主要步骤是：

1. 首先数据按照二叉数来看待，将其下标与二叉树节点对应起来；
2. 构建最大堆（Build-Max-Heap），将堆所有数据重新排序，使其成为最大堆，并且冒出最大数；
3. 堆排序(Heap-Sort)，从最后一个子节点开始遍历，并将根节点与其交换，也就是移除第一个数据的根节点，并做最大堆调整的递归调用，直到排序完成。
4. 堆排序算法执行过程分析：

**待排序数组：**

7, 11, 9, 10, 12, 13, 8

0 1, 2, 3, 4, 5, 6

堆与数组关系

**7(0)**

**/ \**

**11(1) 13(2)**

**/ \ / \**

**10(3) 12(4) 9(5) 8(6)**

遍历父节点，构建大顶堆，取出最大数，得到结果如下：

**13(0)**

**/ \**

**12(1) 9(2)**

**/ \ / \**

**10(3) 11(4) 7(5) 8(6)**

从子节点开始排序，置换顶根节点与子节点，并继续构建大顶堆：

**8(0)**

**/ \**

**12(1) 9(2)**

**/ \ / \**

**10(3) 11(4) 7(5) 13(6)**

子节点排序完成，则堆排序完成，结果如下：

**7(0)**

**/ \**

**8(1) 9(2)**

**/ \ / \**

**10(3) 11(4) 12(5) 13(6)**

1. 堆排序算法代码如下：

// 根据完全二叉树结构

const getParent = (i) => Math.floor((i - 1) / 2)

const getLeft = (i) => 2 \* i + 1

const getRight = (i) => 2 \* i + 2

/\*\*

\* @param {Array<number>} arr

\* @param {number} idx - the index of element

\* @param {number} size - the array's length

\* 让单个堆保持大顶堆特性

\*/

function maxHeapify(arr, parent, size) {

let max = parent

const left = getLeft(parent)

const right = getRight(parent)

if (left < size && arr[left] > arr[max]) {

max = left

}

if (right < size && arr[right] > arr[max]) {

max = right

}

console.log('parent:', parent, 'left:', left, 'right:', right, 'max:', max, 'size:', size)

if (max !== parent) {

// 如果当前不是最大的位置，则进行交换

[arr[parent], arr[max]] = [arr[max], arr[parent]]

// 继续递归执行

maxHeapify(arr, max, size)

}

}

// 非递归构建大顶堆

function maxHeapify(arr, idx, size) {

let current = arr[idx]

let child = getLeft(idx)

// 从当前位置的左节点开始遍历

for(; child < size;) {

console.log('current=', current, ' idx=', idx, ' child=', child, ' size=', size)

// 如果左节点小于右节点且小于总长度，则指向右节点

if(child + 1 < size && arr[child] < arr[child + 1]) {

child++

}

if(arr[child] > current) {

// 如果子节点大于父节点，将子节点的值赋给父节点

arr[idx] = arr[child]

// 当前节点指向该子节点，继续循环

idx = child

} else {

// 子节点小于父节点则跳出循环

break

}

// 遍历子树父节点

child = getLeft(idx)

}

// 赋值为父节点的值

arr[idx] = current

}

function heapSort(arr) {

console.time('heapSort')

const len = arr.length

// 最底层的父节点

let parent = getParent(len) - 1

// 最底层的子节点

let child = len - 1

// 从最后的父节点开始遍历，把最大的那个父节点冒出到堆顶

while (parent >= 0) {

maxHeapify(arr, parent, len)

console.warn('parent sort:', parent, arr)

parent--

}

// 从子节点往上构建大顶堆

while (child > 0) {

// 将顶端的父节点与当前子节点互换

[arr[0], arr[child]] = [arr[child], arr[0]]

// 自最底层往上遍历

maxHeapify(arr, 0, child)

console.warn('child sort:', child, arr);

child--

}

console.timeEnd('heapSort')

return arr

}

const arr = [7, 11, 9, 10, 12, 13, 8]

console.log('sorted:', heapSort(arr))