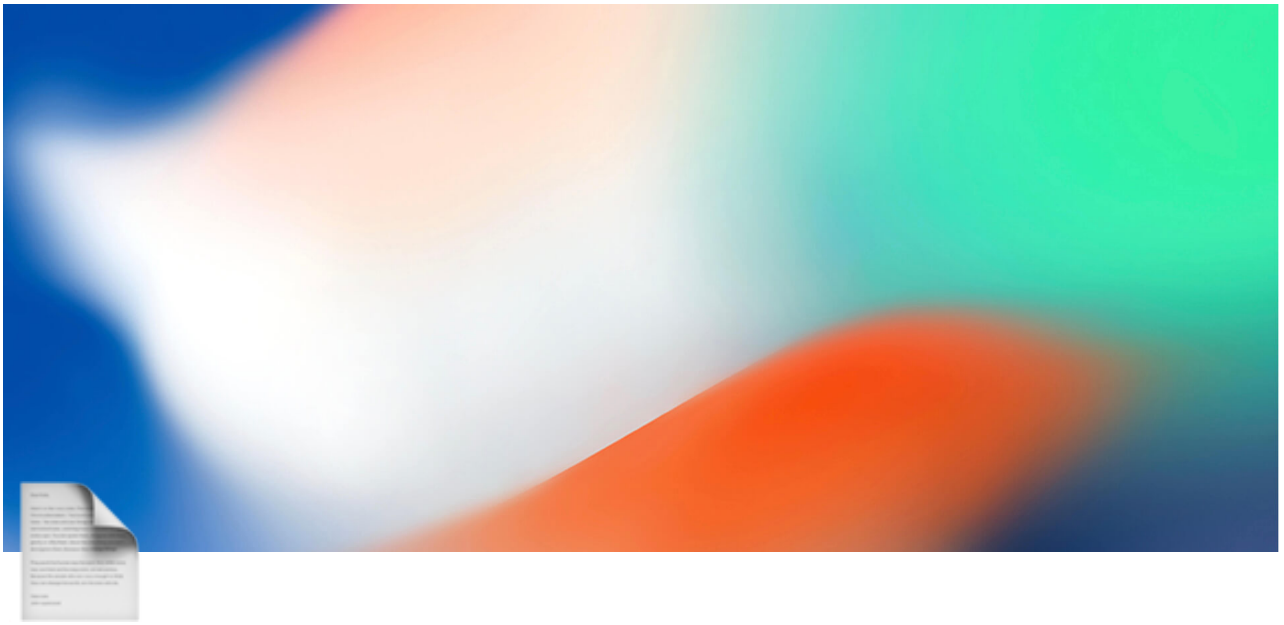


5-1 实现任务调度算法

思考

给你一个数字数组，找出最小的数字，怎么解?

Array.sort ？遍历比较找出最小值?

如果这个数组是动态的，每次我都要找最小值，找到之后就从数组里删除这个元素，然后下次还想找最小值，怎么整。并且这个过程中，还会不断有新的数字插入数组。

Array.sort ？

可是数组是动态的，每次 sort，但是我只要最小值，你浪费那么多时间把第二和第一万都排那么准确，不觉得在浪费时间吗?

算法拓展

对算法感兴趣的同学，可以来 LeetCode 做下相关算法题：LeetCode 703. 数据流中的第 K 大元素

最小堆的前置知识

了解最小堆之前，先来熟悉三个基本数据结构的概念：

二叉树

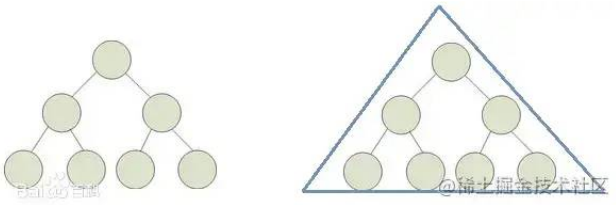
是指树中节点的度不大于 2 的有序树，它是一种最简单且最重要的树。

满二叉树

除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树。

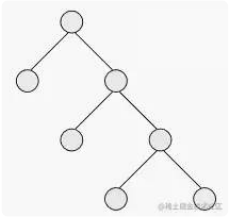
从图形形态上看，满二叉树外观上是一个三角形。

如果一个二叉树的层数为 K，且结点总数是(2^k) -1 ，则它就是满二叉树。



注意： 关于满二叉树定义这里，国内外定义有分歧，本文采用的是国内定义。满二叉树英文是 Full Binary Tree，是指所有的节点的度只能是 0 或者 2。

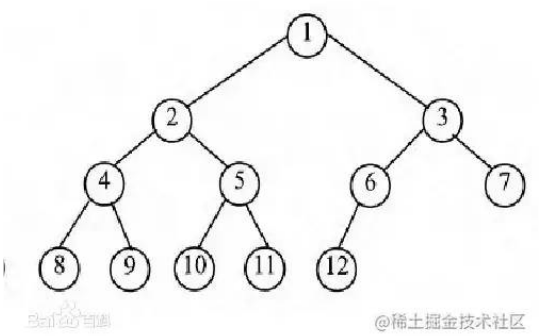
如下图，国外也认为是 Full Binary Tree：



而对于我们本文所说的满二叉树，国外的概念叫完美二叉树。

完全二叉树

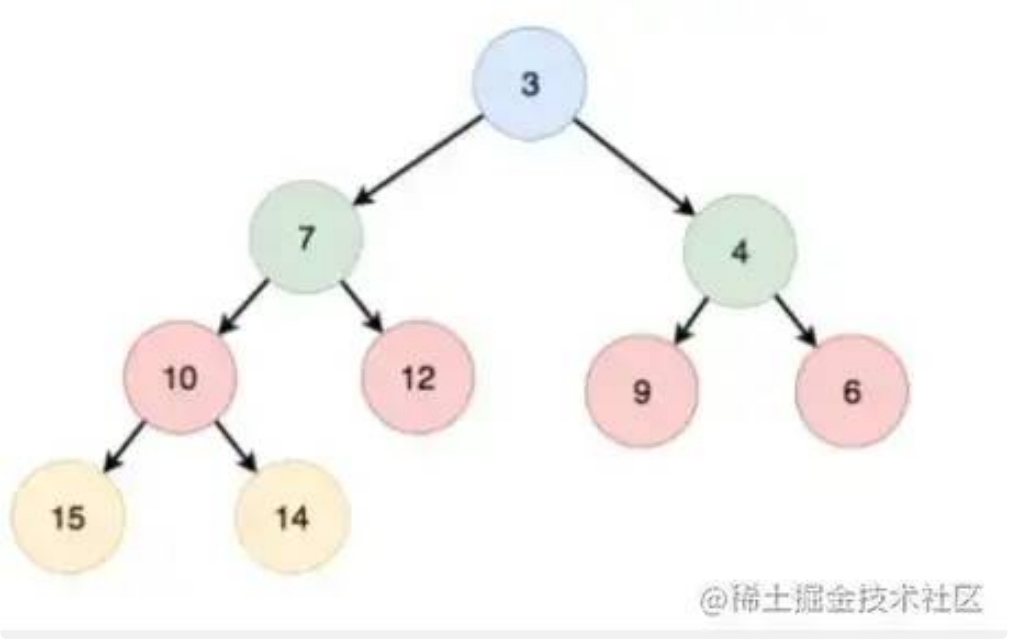
一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i（1≤i≤n）的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。叶子结点只可能在最大的两层出现。



最小堆

是一种经过排序的完全二叉树，其中任一非终端节点的数据值均不大于其左子节点和右子节点的值。

小顶堆。



如对于上面这个最小堆来说，经过观察，对应的深度与数组下标分别是：

数组	3	7	4	10	12	9	6	15	14
深度（depth）	1	2	2	3	3	3	3	4	4
数组下标（index）	0	1	2	3	4	5	6	7	8

经过观察，发现父子节点下标关系如下：

根据子节点下标推算父节点下标：**parentIndex = (childIndex − 1) >>> 1**

根据父节点下标推算子节点下标：

leftIndex = (index +1)*2 – 1,

rightIndex = leftIndex + 1

至此，我们就可以尝试去实现最小堆的增(push)删(pop)查(peek)函数了：

最小堆算法实现

peek

peek，瞄一下嘛，即获取最小堆的堆顶值。

```JavaScript
export function peek<T extends Node>(heap: Heap<T>): T | null {
  return heap.length === 0 ? null : heap[0];
}
```

push

往最小堆中添加一个元素，因为 taskQueue 本身已经是最小堆，并且是数组存储，这时候为了尽可能多的复用原先的结构，我们可以先把新元素插入数组尾部，然后从下往上调整最小堆：

```JavaScript
export function push<T extends Node>(heap: Heap<T>, node: T): void {
  const index = heap.length;
  heap.push(node);
  siftUp(heap, node, index);
}
```

怎么从下往上调整呢？因为最小堆的典型特点就是父节点比左右子节点都小，那这时候除了尾部元素，其他都是满足这个特点的。这个时候我们只需要调整尾部元素以及和尾部元素的祖先就可以了，一直往上调整，直到不再需要调整为止。代码如下：

```JavaScript
function siftUp<T extends Node>(heap: Heap<T>, node: T, i: number): vo
  let index = i;
  while (index > 0) {
    const parentIndex = (index - 1) >>> 1;
    const parent = heap[parentIndex];
```

```
    if (compare(parent, node) > 0) {
      // The parent is larger. Swap positions.
      heap[parentIndex] = node;
      heap[index] = parent;
      index = parentIndex;
    } else {
      // The parent is smaller. Exit.
      return;
    }
  }
}
```

pop

删除堆顶元素。即 React 一个任务执行完了，那么肯定要把这个任务从任务池 taskQueue 中删除。问题来了，怎么删除呢，堆顶元素其实就是 taskQueue[0]，这个位置我们肯定还是要用的，并且和 push 一样，为了尽可能复用原先的最小堆结构，我们可以采取一个办法：把最后一个元素覆盖堆顶元素，然后从堆顶往下调整最小堆。

JavaScript

```
export function pop<T extends Node>(heap: Heap<T>): T | null {
  if (heap.length === 0) {
    return null;
  }
  const first = heap[0];
  const last = heap.pop();
  if (last !== first) {
    if (last) {
      heap[0] = last;
      siftDown(heap, last, 0);
    }
  }
  return first;
}
```

关于往下调整，其实就是检查每个子堆的结构，确保最小值在父节点，不满足就交换父与左或者父与右，代码如下，我加了尽可能多的详细注释：

JavaScript

```
function siftDown<T extends Node>(heap: Heap<T>, node: T, i: number):
  let index = i;
  const length = heap.length;
  const halfLength = length >>> 1;
  while (index < halfLength) {
    const leftIndex = (index + 1) * 2 - 1;
    const left = heap[leftIndex];
    const rightIndex = leftIndex + 1;
    const right = heap[rightIndex];

    // If the left or right node is smaller, swap with the smaller of
    if (compare(left, node) < 0) {
      if (rightIndex < length && compare(right, left) < 0) {
        heap[index] = right;
        heap[rightIndex] = node;
        index = rightIndex;
      } else {
        heap[index] = left;
        heap[leftIndex] = node;
        index = leftIndex;
      }
    } else if (rightIndex < length && compare(right, node) < 0) {
      heap[index] = right;
      heap[rightIndex] = node;
      index = rightIndex;
    } else {
      // Neither child is smaller. Exit.
      return;
    }
  }
}
```