【第六课】堆与优先队列

小顶堆实现

```
// 助教小杨和Web1~13班，大家共同的杰作。
class Heap {
    constructor(data) {
        this.data = data;
        this.compartor = (a, b) => a - b;
        this.heapify();
    }

    size() {
        return this.data.length;
    }

    heapify() {
        if (this.size() < 2) {
            return;
        }
        for (let i = 1; i < this.size(); i++) {
            this.bubbleUp(i);
        }
    }

    peek() {
        if (!this.size()) return null;
        return this.data[0];
    }

    offer(val) {
        this.data.push(val);
        this.bubbleUp(this.size() - 1);
    }

    poll() {
        if (!this.size()) return null;
        let res = this.data[0];
        this.data[0] = this.data.pop();
        if (this.size()) {
            this.bubbleDown(0);
        }
        return res;
    }

    swap(i, j) {
        if (i === j) {
            return;
        }
        [this.data[i], this.data[j]] = [this.data[j], this.data[i]];
    }

    bubbleUp(index) {
        // 向上调整，我们最高就要调整到0号位置
        while (index) {
            //获取到当前节点的父节点，
            const parenIndex = (index - 1) >> 1;
            // const parenIndex = Math.floor((index - 1) / 2);
            // const parenIndex = (index - 1) / 2 | 0;
            //比较父节点的值和我们当前的值哪个小。
            if (this.compartor(this.data[index], this.data[parenIndex]) < 0) {
                //if   交换父节点和子节点
                this.swap(index, parenIndex);
                // index 向上走一步，进行下一次交换
                index = parenIndex;
```

```
            } else {
                //防止死循环。
                break;
            }
        }
    }
    bubbleDown(index) {
        //我们要获取到最大的下标，保证不会交换出界。
        let lastIndex = this.size() - 1;
        while (index < lastIndex) {
            //获取左右儿子的下标
            let leftIndex = index * 2 + 1;
            let rightIndex = index * 2 + 2;
            // 待交换节点
            let findIndex = index;
            if (leftIndex <= lastIndex
                && this.compartor(this.data[leftIndex], this.data[findIndex]) < 0) {
                findIndex = leftIndex;
            }
            if (rightIndex <= lastIndex && this.compartor(this.data[rightIndex], this.data[findIndex]) < 0) {
                findIndex = rightIndex;
            }
            if (index !== findIndex) {
                this.swap(index, findIndex);
                index = findIndex;
            } else {
                break;
            }
        }
    }
}
let arr = [8, 7, 6, 9];
let minHeap = new Heap(arr);
console.log(minHeap.poll());
console.log(minHeap.data);
```

剑指 Offer 40. 最小的k个数

我们可以维护一个小根堆用来给所有元素排序。排序后堆中的前K个元素就是我们需要的元素。

```
/**
 * @param {number[]} arr
 * @param {number} k
 * @return {number[]}
 */
var getLeastNumbers = function (arr, k) {
    let len = arr.length;
    let res = [];
    if(k===0)return [];
    if(k===len)return arr;
    buildHeap(arr);
    for (let i = 1; i <= k; i++) {
        res.push(arr[0]);
        swap(arr, 0, len - i);
        heapAdjust(arr, 0, len - i);
    }
    return res;
};
var buildHeap = function (arr) {
    let len = arr.length;
    for (let i = Math.floor(len / 2); i >= 0; i--) {
        heapAdjust(arr, i, len);
    }
}

function swap(arr, i, child) {
```

```
    if (i === child) return;
    arr[i] = arr[child] + arr[i];
    arr[child] = arr[i] - arr[child];
    arr[i] = arr[i] - arr[child];
}

function heapAdjust(arr, i, len) {
    let child = i * 2 + 1;
    while (child < len) {
        if (child + 1 < len && arr[child] > arr[child + 1]) {
            child = child + 1;
        }
        if (arr[child] < arr[i]) {
            swap(arr, i, child);
            i = child;
            child = i * 2 + 1;
        } else {
            break;
        }
    }
}
```

1046. 最后一块石头的重量

我们可以维护一个大根堆，然后每次取出堆顶的元素，两两相减，将结果再加入到堆中，直到堆中的元素小于两个。

```
var lastStoneWeight = function (stones) {
    const maxHeap = new MaxPriorityQueue();
    for (let i = 0; i < stones.length; i++) {
        maxHeap.enqueue('x', stones[i]);
    }
    while (maxHeap.size() > 1) {
        const a = maxHeap.dequeue()['priority'];
        const b = maxHeap.dequeue()['priority'];
        if (a > b) {
            maxHeap.enqueue('x', a - b);
        }
    }

    return maxHeap.isEmpty() ? 0 : maxHeap.dequeue()['priority'];
};
```

703. 数据流中的第 K 大元素

我们可以维护一个大小为K的小根堆，用来存储前K大的元素。然后将数据流中的数据加入到小根堆中进行调整，返回堆顶的元素。

```


/**
 * @param {number} k
 * @param {number[]} num
 */
var KthLargest = function(k, nums) {
    this.k = k;
    this.heap = new MinHeap();
    for(n of nums){
        this.add(n);
    }
};
```

```
/**
 * @param {number} val
 * @return {number}
 */
KthLargest.prototype.add = function(val) {
    this.heap.offer(val);
    if(this.heap.size()>this.k){
        this.heap.poll();
    }
    return this.heap.peek();
};
class MinHeap {
    constructor(data = []) {
        this.data = data;
        this.comparator = (a, b) => a - b;
        this.heapify();
    }

    heapify() {
        if (this.size() < 2) return;
        for (let i = 1; i < this.size(); i++) {
            this.bubbleUp(i);
        }
    }

    peek() {
        if (this.size() === 0) return null;
        return this.data[0];
    }

    offer(value) {
        this.data.push(value);
        this.bubbleUp(this.size() - 1);
    }

    poll() {
        if (this.size() === 0) {
            return null;
        }
        const result = this.data[0];
        const last = this.data.pop();
        if (this.size() !== 0) {
            this.data[0] = last;
            this.bubbleDown(0);
        }
        return result;
    }

    bubbleUp(index) {
        while (index > 0) {
            const parentIndex = (index - 1) >> 1;
            if (this.comparator(this.data[index], this.data[parentIndex]) < 0) {
                this.swap(index, parentIndex);
                index = parentIndex;
            } else {
                break;
            }
        }
    }

    bubbleDown(index) {
        const lastIndex = this.size() - 1;
        while (true) {
            const leftIndex = index * 2 + 1;
            const rightIndex = index * 2 + 2;
            let findIndex = index;
            if (
                leftIndex <= lastIndex &&
                this.comparator(this.data[leftIndex], this.data[findIndex]) < 0
            ) {
                findIndex = leftIndex;
```

```
            }
            if (
                rightIndex <= lastIndex &&
                this.comparator(this.data[rightIndex], this.data[findIndex]) < 0
            ) {
                findIndex = rightIndex;
            }
            if (index !== findIndex) {
                this.swap(index, findIndex);
                index = findIndex;
            } else {
                break;
            }
        }
    }

    swap(index1, index2) {
        [this.data[index1], this.data[index2]] = [this.data[index2], this.data[index1]];
    }

    size() {
        return this.data.length;
    }
}
```

373. 查找和最小的K对数字

我们固定住一个数组的元素，然后去遍历另一个数组求和。然后维护一个K大小的大根堆，将每个组合与堆顶元素进行比较，如果小于堆顶元素就加入到大根堆中，如果大于堆顶元素，由于数组是有序的，我们终止当前的循环，进行下一次循环。如果堆的大小超过了K就进行将堆顶元素弹出。

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @param {number} k
 * @return {number[][]}
 */
var kSmallestPairs = function (nums1, nums2, k) {
    const heap = [];
    for (let i = 0; i < nums1.length; i++) {
        for (let j = 0; j < nums2.length; j++) {
            if (heap.length < k) {
                heap.push([nums1[i], nums2[j]]);
                shiftUp(heap, heap.length - 1);
            } else if ((nums1[i] + nums2[j]) <= sum(heap[0])) {
                heap[0] = [nums1[i], nums2[j]];
                shiftDown(heap, 0);
            }
        }
    }
    return heap.sort((a, b) => (a[0] + a[1]) - (b[0] + b[1]));
};

function swap(heap, index, parent) {
    [heap[index], heap[parent]] = [heap[parent], heap[index]]
}

function shiftUp(heap, i) {
    const parent = (i - 1) / 2 | 0
    if (sum(heap[i]) > sum(heap[parent])) {
        swap(heap, i, parent)
        shiftUp(heap, parent)
    }
}
```

```
function sum(arr) {
    return arr[0] + arr[1];
}

function shiftDown(heap, index) {
    let left = index * 2 + 1;
    if (left >= heap.length) return;
    if (left + 1 < heap.length && sum(heap[left]) < sum(heap[left + 1])) {
        left = left + 1;
    }
    if (sum(heap[index]) <= sum(heap[left])) {
        swap(heap, index, left)
        shiftDown(heap, left)
    }
}
```

215. 数组中的第K个最大元素

我们可以维护一个大小为K的小根堆，用来存储前K大的元素。然后将数组中的数据加入到小根堆中进行调整，最后返回堆顶的元素。

```
var findKthLargest = function (nums, k) {
    let heap = [,], i = 0;
    while (i < k) {
        heap.push(nums[i++]);
    }
    buildHeap(heap, k);

    for (let i = k; i < nums.length; i++) {
        if (heap[1] < nums[i]) {
            heap[1] = nums[i];
            heapify(heap, k, 1);
        }
    }
    return heap[1];
};

function heapify(arr, k, i) {
    while (true) {
        let minIndex = i;
        if (2 * i <= k && arr[2 * i] < arr[i]) {
            minIndex = 2 * i;
        }
        if (2 * i + 1 <= k && arr[2 * i + 1] < arr[minIndex]) {
            minIndex = 2 * i + 1;
        }
        if (minIndex !== i) {
            swap(arr, i, minIndex);
            i = minIndex;
        } else {
            break;
        }
    }
}

var buildHeap = function (arr, k) {
    if (k === 1) return;
    for (let i =Math.floor(k / 2); i >= 1; i--) {
        heapify(arr, k, i);
    }
}
let swap = (arr, i, j) => {
    let temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```

355. 设计推特

我们需要一个大小为10的大根堆，用来存储最新的十条推文。我们需要创建一个推文对象以及一个用户对象。然后我们的关注功能可以通过set来记录当前用户关注的人，同时每个用户要关注自身。每次我们获取推文的时候，我们就将关注列表中的用户的推文取出加入大根堆中，进行调整。当所有推文都加入一次大根堆后，堆中的推文就是我们需要的推文。

```
/**
 * Initialize your data structure here.
 */
var Twitter = function () {
    this.userMap = new Map();
};
/**
 * Compose a new tweet.
 *@param{number} userId
 *@param{number} tweetId
 *@return{void}
 */
Twitter.prototype.postTweet = function (userId, tweetId) {
    if (!this.userMap.has(userId)) {
        this.userMap.set(userId, new User(userId));
    }
    var u = this.userMap.get(userId);
    u.post(tweetId);
};

/**
 * Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be poste
d by users who the user followed or by the user herself. Tweets must be ordered from most recent to least
 recent.
 *@param{number} userId
 *@return{number[]}
 */
Twitter.prototype.getNewsFeed = function (userId) {
    var h = new Heap();
    var res = [], candidates = [];
    if (!this.userMap.has(userId)) {
        return res;
    }
    //获取关注列表，将列表中的推特放入候选推特堆
    for (let ids of this.userMap.get(userId).followed) {
        candidates = candidates.concat(this.userMap.get(ids).tweets);
    }
    // 根据时间调整堆
    h.build(candidates, 'time');
    // 根据时间拿出最新的十条推特
    while (res.length < 10 && h.data.length) {
        res.push(h.deleting('time').tweetId);
    }
    return res;
};

/**
 * Follower follows a followee. If the operation is invalid, it should be a no-op.
 *@param{number} followerId
 *@param{number} followeeId
 *@return{void}
 */
Twitter.prototype.follow = function (followerId, followeeId) {
    if (!this.userMap.has(followerId)) {
        this.userMap.set(followerId, new User(followerId));
    }
```

```
    if (!this.userMap.has(followeeId)) {
        this.userMap.set(followeeId, new User(followeeId));
    }
    this.userMap.get(followerId).follow(followeeId);
};

/**
 * Follower unfollows a followee. If the operation is invalid, it should be a no-op.
 *@param{number} followerId
 *@param{number} followeeId
 *@return{void}
 */
Twitter.prototype.unfollow = function (followerId, followeeId) {
    if (this.userMap.has(followerId)) {
        this.userMap.get(followerId).unfollow(followeeId);
    }
};

/**
 * Your Twitter object will be instantiated and called as such:
 * var obj = new Twitter()
 * obj.postTweet(userId,tweetId)
 * var param_2 = obj.getNewsFeed(userId)
 * obj.follow(followerId,followeeId)
 * obj.unfollow(followerId,followeeId)
 */
//发推时间
var timeStamp= 0;
//创建推文对象
var Tweet = function (tweetId, timeStamp) {
    this.tweetId = tweetId;
    this.time = timeStamp;
};
//创建用户对象
var User = function (userId) {
    //用户Id
    this.id = userId;
    //关注列表
    this.followed = new Set();
    // 发送推文列表
    this.tweets = [];

    this.follow(userId);
};
User.prototype.follow = function (userId) {
    // 注意followed装进去的都是userID
    this.followed.add(userId);
};
User.prototype.unfollow = function (userId) {
    if (userId !== this.id) {
        this.followed.delete(userId);
    }
};

User.prototype.post = function (tweetId) {
    var tweet = new Tweet(tweetId,timeStamp);
timeStamp++;
    // 最新的推文永远在最前面
    this.tweets.unshift(tweet);
};
//创建大根堆
function Heap() {
    this.data = [];
    this.build = build;
    this.insert = insert;
    this.deleting = deleting;
    this.heapSort = heapSort;
}

function build(arr, key) {
    for (var i = 0; i < arr.length; i++) {
```

```
        this.insert(arr[i], key);
    }
}

function insert(val, key) {
    this.data.push(val);
    var idx = this.data.length - 1;
    var fatherIdx =Math.floor((idx - 1) / 2);
    // 构建大根堆的过程：寻找父节点，如果比父节点大就交换，一直到根节点为止
    while (fatherIdx >= 0) {
        if (this.data[idx][key] > this.data[fatherIdx][key]) {
            var temp = this.data[idx];
            this.data[idx] = this.data[fatherIdx];
            this.data[fatherIdx] = temp;
        }
        idx = fatherIdx;
        fatherIdx =Math.floor((idx - 1) / 2);
    }
}

/**
 *删除根节点，并且保持堆数据结构不变（维持大根堆）
*时间复杂度:O(logn)
 *@returns{*}
 */
function deleting(key) {
    if (this.data.length === 1) {
        return this.data.pop();
    }
    var idx = 0;
    var val = this.data[idx];
    // 把最后一个元素翻到根节点上，然后开始从根节点向下遍历保证父节点的值总是大于子节点
    this.data[idx] = this.data.pop();
    while (idx < this.data.length) {
        var left = 2 * idx + 1;
        var right = 2 * idx + 2;
        var select = left;
        // 首先要查找出左右哪个更大
        if (right < this.data.length) {
            select = (this.data[left][key] < this.data[right][key]) ? right : left;
        }
        if (select < this.data.length && this.data[idx][key] < this.data[select][key]) {
            var temp = this.data[idx];
            this.data[idx] = this.data[select];
            this.data[select] = temp;
        }
        idx = select;
    }
    return val;
}

/**
 *堆排序
*/
function heapSort() {
    let res = [];
    while (this.data.length > 0) {
        res.unshift(this.deleting());
    }
    return res;
}
```

692. 前K个高频单词

我们可以利用Map来计算每个单词出现的次数，然后维护一个大小为K的小根堆，将单词按次数加入到小根堆中进行调整，如果次数相同，就比较单词。最后堆中剩余的单词就是我们需要的单词。

```
var topKFrequent = function (words, k) {
    let map = new Map();
    let heap = [];
    words.forEach(item => {
        map.has(item) ? map.set(item, map.get(item) + 1) : map.set(item, 1);
    })
    let i = 0;
    map.forEach((value, key) => {
        if (i < k) {
            heap.push([key, value]);
            i === k - 1 && buildHeap(map, heap, k);
        } else if (value > map.get(heap[0][0]) || (value === map.get(heap[0][0]) && key < heap[0][0])) {
            heap[0] = [key, value];
            heapify(map, heap, k, 0);
        }
        i++;
    });
    let temp = heap.sort((a, b) => {
        if (a[1] > b[1]) {
            return -1;
        } else if (a[1] < b[1]) {
            return 1;
        } else {
            if (a[0] > b[0]) {
                return 1;
            } else if (a[0] < b[0]) {
                return -1;
            }
        }
    });
    let res = [];
    temp.forEach(item => {
        res.push(item[0]);
    })
    return res;
};
var buildHeap = function (map, arr, len) {
    for (let i =Math.floor(len / 2); i >= 0; i--) {
        heapify(map, arr, len, i);
    }
}
var heapify = function (map, arr, len, i) {
    let l = 2 * i + 1, r = 2 * i + 2, minIndex = i;
    // 次数小或者相等情况排序靠前的置于小堆顶
    if (l < len
        && (map.get(arr[l][0]) < map.get(arr[minIndex][0])
            || (map.get(arr[l][0]) === map.get(arr[minIndex][0])
                && arr[l][0] > arr[minIndex][0]))) {
        minIndex = l;
    }
    if (r < len
        && (map.get(arr[r][0]) < map.get(arr[minIndex][0])
            || (map.get(arr[r][0]) === map.get(arr[minIndex][0])
                && arr[r][0] > arr[minIndex][0]))) {
        minIndex = r
    }
    if (minIndex !== i) {
        swap(arr, minIndex, i);
        heapify(map, arr, len, minIndex);
    }
}
var swap = function (arr, i, j) {
    const temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```