Stack, Queues, Priority Queues & Heaps Lecture 4

Sunday, 11 August 2024 10:03 AM

https://leetcode.com/problems/kth-largest-element-in-a-stream/

pair <int, int>

pair <int, pair<int, int>>

Priority queues

priority_queue < T > pq; ← Max heap

int

Min heap } use as

by entering negative values

arr = [4, 5, 6, 1, 3, 2, 7, 4] k = 3

k^{th} largest = 5

Top k queries

① arr [4, 5, 6, 1, 3, 2, 7, 4] → O(n log n) for sorting.

↓ sort

[1, 2, 3, 4, 4, 5, 6, 7]

for every query

Time :- O(n)

space :- O(n)

add :-

add(5) [1, 2, 3, 4, 4, 5, 5, 6, 7]

② Just keep k elements. k = 3

[4, 5, 6, 1, 3, 2, 7, 4]

↓ sort } (n log n)

[1, 2, 3, 4, 4, 5, 6, 7] } O(k)

Top 3

[7, 6, 5]

for every query.

space O(k) ← Time O(k)

add(5) [7, 6, 5]

add(8) [8, 7, 6]

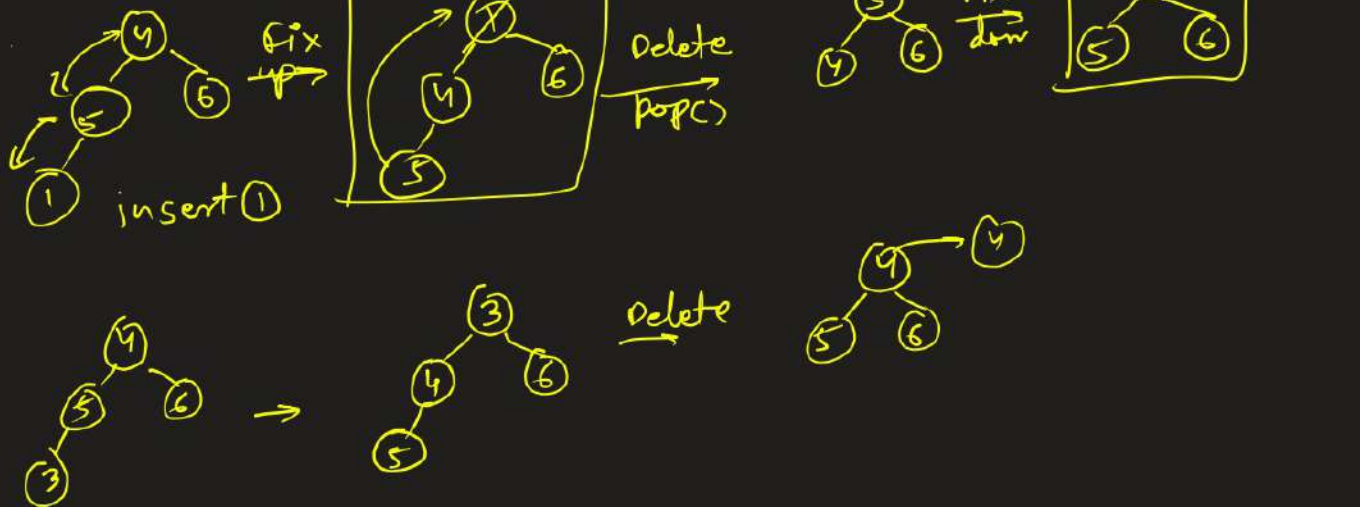
O(k)

O(k)

③ heaps

[4, 5, 6, 1, 3, 2, 7, 4]

min heap of size k



Keep K largest elements in a min heap.

add (_) → add to heap insert (log k)

if heap ≥ k → delete an element pop() (log k)

top() → kth largest element O(1)

Time → log(k) Space → O(k)

```
class KthLargest {
public:
    priority_queue<int> pq;
    int k;
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        for(auto n: nums) {
            pq.push(-n); // Min-Heap
            if(pq.size()>k)
                pq.pop();
        }
    }

    int add(int val) {
        pq.push(-val);
        if(pq.size() > k)
            pq.pop();
        return -pq.top();
    }
};
```

https://leetcode.com/problems/top-k-frequent-elements/

1, 1, 1, 2, 2, 3, 3, 2, 4, 4, 1, 5

Freq ele

1 → 4 : <4, 1>
2 → 3 : <3, 2>
3 → 2 : <2, 3>
4 → 2 : <2, 4>
5 → 1 : <1, 5>

time:- Space O(D) Time O(n)

min-heap → <3, 2>
<4, 1>

O(k)
O(D log k)

Space = O(k)

K = 2 Time → O(D) Space = O(D)

O(d)
# distinct

<4, 1>
<3, 2> <2, 3>
<2, 4> <1, 5>

max-heap

<3, 2>
<2, 4> <2, 3>
<1, 5>

<3, 2>

```cpp
// Overall: T: O(n+k log d), S: O(d+k)
    vector<int> topKFrequent(vector<int>& nums, int k) {
        // Frequency map
        // T: O(n), S: O(d)
        unordered_map<int, int> um;
        for(auto n: nums)
            um[n]++;
        // Vector containing (freq, ele) pairs
        // T: O(d), S: O(d)
        vector<pair<int, int>> vec;
        for(auto &[k, v]: um)
            vec.push_back({v, k});

        // Build heap (Heapify)
        // T: O(d), S: O(d)
        priority_queue<pair<int, int>> pq(vec.begin(), vec.end());
        // Take the top k elements
        // T: O(k log d), S: O(k)
        vector<int> ans;
        for(int i=0; i<k; i++) {
            ans.push_back(pq.top().second);
            pq.pop();
        }
        return ans;
    }
```

Comp:-

pq < pair<int, int> >
ele freq.

<ele,freq> <ele,freq>
comp (pair1, pair2)
p1 p2

return p1.second < p2.second

greater <int> → pq → min heap

greater (a, b)

return a > b

True → first > second } → min_heap
False → First < second }

True → First < second } → max_heap
False → First > second }

comparator → override < operator

a < b

→ True
False

sort (, greater<int>)

↳ a > b → True

pq < > → max heap a < b

pq < greater<int>> pq a < b ↳ a > b

pq < pair<int, int>, comp > pq

```cpp
class comp {
public:
    bool operator()(pair<int, int> p1, pair<int, int> p2) {
        return p1.second < p2.second; // max heap on second values
    }
};
class Solution {
public:
    // Overall: T: O(n+k log d), S: O(d+k)
    vector<int> topKFrequent(vector<int>& nums, int k) {
        // Frequency map
        // T: O(n), S: O(d)
        unordered_map<int, int> um;
        for(auto n: nums)
            um[n]++;

        // Build heap (Heapify)
        // T: O(d), S: O(d)
        priority_queue<pair<int, int>, vector<pair<int, int>>, comp>
pq(um.begin(), um.end());
        // Take the top k elements
        // T: O(k log d), S: O(k)
        vector<int> ans;
        for(int i=0; i<k; i++) {
            ans.push_back(pq.top().first);
            pq.pop();
        }
        return ans;
    }
};
```

```cpp
class comp {
public:
    bool operator()(pair<int, int> p1, pair<int, int> p2) {
        return p1.second > p2.second; // min heap on second values
    }
};
class Solution {
public:
    // Overall: T: O(n+k log d), S: O(d+k)
    vector<int> topKFrequent(vector<int>& nums, int k) {
        // Frequency map
        // T: O(n), S: O(d)
        unordered_map<int, int> um;
        for(auto n: nums)
            um[n]++;

        // Build heap (Heapify)
        // T: O(klogk), S: O(k)
        priority_queue<pair<int, int>, vector<pair<int, int>>, comp> pq;
        for(auto &[key, val]: um) {
            pq.push({key, val});
            if(pq.size() > k)
                pq.pop();
        }
        // Take the top k elements
        // T: O(k log d), S: O(k)
        vector<int> ans;
        for(int i=0; i<k; i++) {
            ans.push_back(pq.top().first);
            pq.pop();
        }
        return ans;
    }
};
```

https://leetcode.com/problems/find-median-from-data-stream/

Brute:- Vector <int> arr;

push(k) arr.push_back(k); O(1) ✓

find_median() →
↳ Sort the array. (n log n)
middle element O(1) } ✓

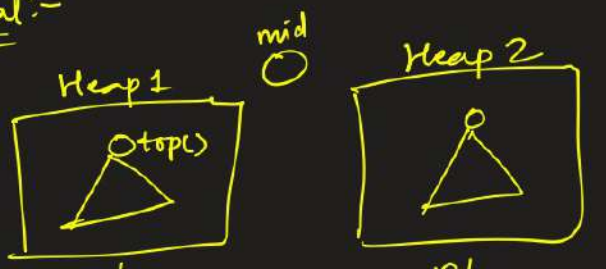
Slightly optimized:-

Keep the sorted array. [1, 3, 5, 7] →

push(k) → push(6) [1, 3, 5, 6, 7] O(n) } ✓

find_median() → middle O(1) } ✓

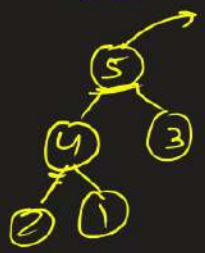
Optimal:-

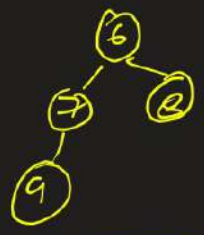


[3, 4, 1, 2, 5, 6, 7, 8, 9]

n/2
max-heap
≤ mid

n/2
min-heap
> mid





find_median — O(1)

```
if (h1.size() == h2.size())          // O(1) O(1)
    return (h1.top() + h2.top()) / 2; // O(1) O(1)
else return h1.top();                // O(1)
```

add(k) →

- → Insert k in h1
- → Find the top of h1 → temp
- → pop the top of h1
- → Insert temp in h2
- → If size of h2 > size of h1
  - → Find the top of h2 → temp
  - → pop the top of h2
  - → insert temp in h1

log(n/2) + log(n/2) + log(n/2)

≤ log(n)

≤ O(n/2) + O(n/2)

= O(n)

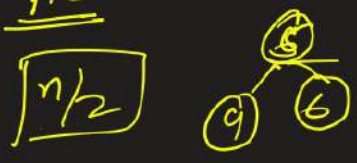
[3, 9, 4, 6, 5, 2, 7, 1, 8]

max heap

H1

min heap

H2



temp = ⑤
max element in the left half

size of $H_2 \geq H_1$
temp = insert in H_1

```
class MedianFinder {
public:
    priority_queue<int> h1, h2; // h1: maxheap, h2: minheap
    MedianFinder() {
        while(!h1.empty()) h1.pop();
        while(!h2.empty()) h2.pop();
    }

    void addNum(int num) {
        h1.push(num); // max heap
        int temp = h1.top();
        h1.pop();
        h2.push(-temp); // min heap
        if(h1.size() != h2.size()) {
            temp = -h2.top();
            h2.pop();
            h1.push(temp);
        }
    }

    double findMedian() {
        if(h1.size() == h2.size())
            return (double(h1.top())-h2.top())/2.0;
        return h1.top();
    }
};
```

https://leetcode.com/problems/merge-k-sorted-lists/