Data Structure

7. Heap

Handwritten by pankaj kumar

Heaps

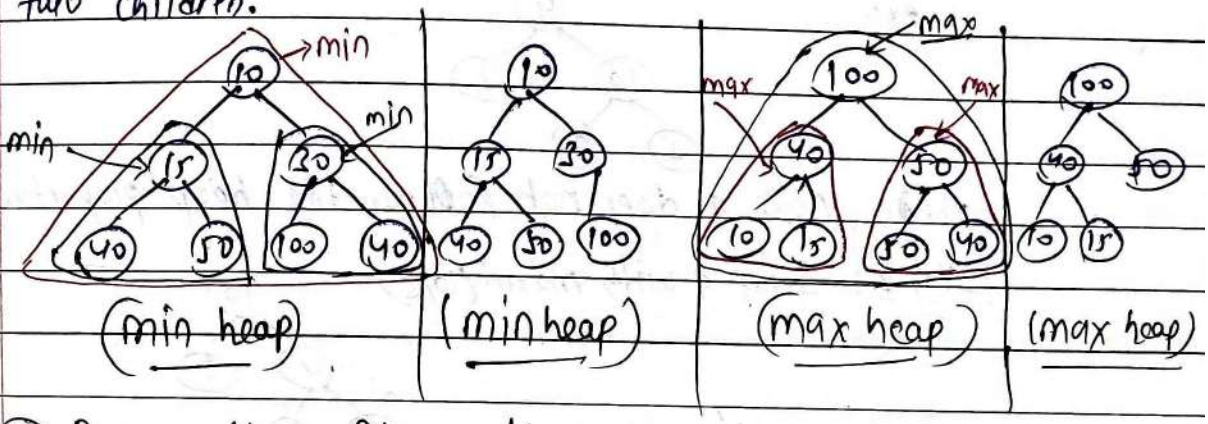
Date
Page No. 366.

① Introduction

- A Heap is a Tree-based ds. with condition
  - A Heap is a complete tree (A level are completely filled except possibly the last level and the last level has all keys as left as possible)
  - A Heap is either min Heap or max Heap. In a min-Heap, the key at root must be minimum among all keys present in the heap. The same property must be true for all nodes.

✱ Binary Heap:-

A Binary Heap is a Heap where each node can have at most two children.



✱ Representing Binary Heaps:-

- since a Binary heap is a complete Binary Tree, it can be easily represented using array.
  - The root element will be at Arr[0].
  - For any i^{th} node, i.e., Arr[i]:

    Arr[(i-1)/2] Return the parent node

    Arr[(2*i)+1] Return the left child node

    Arr[(2*i)+2] Return the right child node

Getting maxm element: the maxm element in a max heap is always at root i.e Arr[0], can be accessed in O(1) time.

Getting minm element: the minm element in a min heap is always at root i.e Arr[0], can be accessed in O(1) time.

ex:- [10|15|30|40|50|100|40]
(min heap)

ex:- [100|40|50|10|15|50|40]
(max heap)

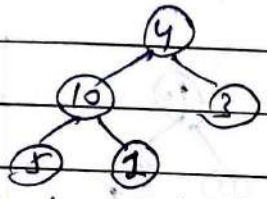
(*) Heapifying an element :-

- Generally, on inserting a new element onto a heap, it does not satisfy the property of Heap. the process of placing the element at the correct location so that it satisfies the heap property is known as heapify.

Heapifying in a max Heap:-

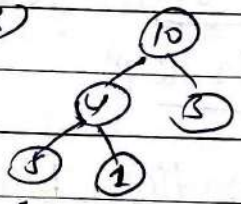
the max-heap says that every nodes' value must be greater than the values of its children nodes. so, to heapify a particular node swap the value of node with the maxm value of its children nodes and continue until become heap.

ex:-

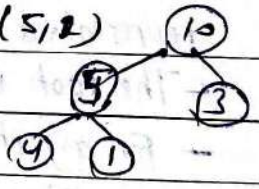


- here, node 4 does not follow the heap property. let's heapify

Step 1:- swap 4 with maxm (10,3)



Step 2:- again node 4 does not heap property. swap 4 with maximum of its new children max(5,2)



node 4 is now heapified successfully.

Note:- Time complexity to heapify a single node is $O(h)$, here n = total no. of element in heap. or, $O(\log n)$

(2) Insertion & Deletion in Heaps :-

(*) Deletion in Heap :-

- Given a Binary heap & an element present in the given Heap. The task is to delete that element from heaps.
- The standard deletion is to delete the max/min element present at root of heap.

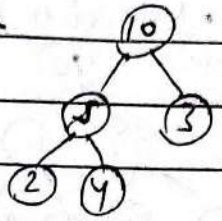
(O(log n))

(i) Process of root Deletion: / Extract max, Extract min (O(1))

since deleting from intermediate position is costly from a heap, so we can simply replace with last element & delete last.

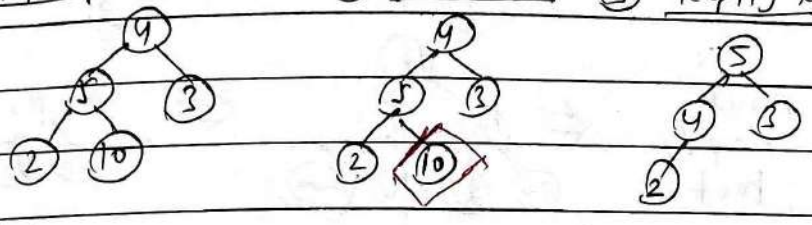
→ Replace the root by last
→ Delete last from the heap
→ since now last element position is changed so heapify them

Eg:-



delete: 10

① Replace with last ② Delete last ③ heapify Root(4)



Code:-

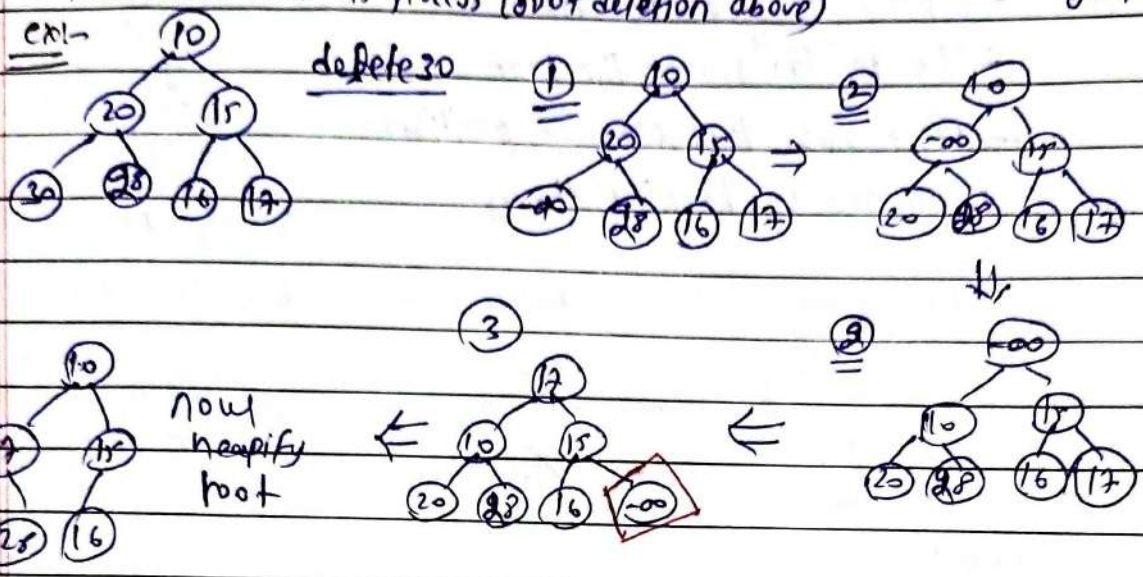
```
void heapify(int arr[], int n, int i){        // Heapify
    int largest = i;
    int l = 2*i+1;
    int r = 2*i+2;
    if (l<n && arr[l] > arr[largest])
        largest = l;
    if (r<n && arr[r] > arr[largest])
        largest = r;
    if (largest != i){swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
void deleteRoot(int arr[], int &n)        // delete root.
    int lastElement = arr[n-1];
    arr[0] = lastElement;
    n = n-1;
    heapify(arr, n, 0);
}
```

O(h) = O(log n)

(iii) delete non-root element at given index in min heap

- Replace the element to be delete with minus infite.
- keep swapping the minus infinite value with parent until it reaches root
- Now delete root as process (root deletion above)

ex:-



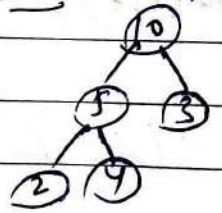
(*) Insertion in Heaps:-

- Insert a given element to the given heap maintaining the properties of Heap.

process of insertion:

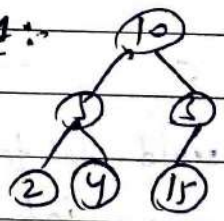
- increase the heap size by 1. to store new element.
- insert new element at end of the heap.
- this new element may distort properties of Heap for its parent, so heapify this newly inserted element following a bottom up approach.

Ex:- suppose given max-heap as:-



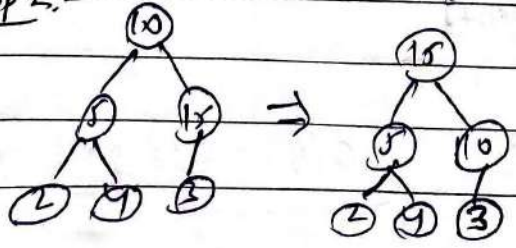
new element = 15

Step 1:-



heapify 15 in bottom up.

Step 2:-



[n=size, key=element insert

Code:-

② void heapify (int arr[], int n, int i){
    int parent = (i-1)/2
    if (parent >= 0){
        if (arr[i] > arr[parent]){
            swap (arr[i], arr[parent]);
            heapify (arr, n, parent);
        }
    }
}

① void insertNode (int arr[], int &n, int key){
    n = n+1;
    arr[n-1] = key;
    heapify (arr, n, n-1);
}

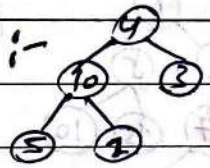
TC: O(log n)

③ Building Heap :-

- Given N element the task is to build a Binary Heap from the given array. Heap can be max Heap or min Heap.

Ex:- given element are [4, 10, 3, 5, 1].

complete binary tree will be :-



(i) root is at index '0'
(ii) left child of i is at (2*i+1) idx.
(iii) right is at (2*i+2) idx.
(iv) parent of i is at (i-1)/2

* process to make Max-Heap :-

- The idea is to make heapify the complete binary tree formed from the array in reverse level order following top down approach.
- that is first heapify the last node in level order traversal of tree, then heapify the second last node & so one.

Time complexity :- Heapy a single node take O(log N), so heapify total N node will take O(N*log N) time.

① • max^m no. of nodes at height $h = \lceil \frac{N}{2^{h+1}} \rceil$

∴ total time for level h to heapify $= \lceil \frac{N}{2^{h+1}} \rceil \cdot O(h)$

∴ for all node at each level $T(N) = \sum_{h=0}^{\log N} \lceil \frac{N}{2^{h+1}} \rceil \cdot O(h) = O(N)$

⇒ Proof of O(N) time.
we have to go h element above.
372.

Optimized approach :- $(T(N) : O(N))$

- the leaf node need not to be heapified as they already follow the heap property. Also, the array representation of complete binary tree contains the level order traversal.
- So the idea is to find the position of the last non-leaf node and perform the heapify operation of each non-leaf node in reverse level order.

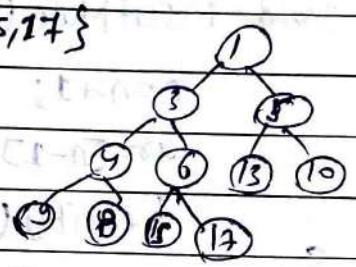
last non-leaf node = parent of last node $(n-1)$

$= ((n-1)-1)/2 = (n-2)/2$

Ex:-

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

corresponding complete Binary Tree is



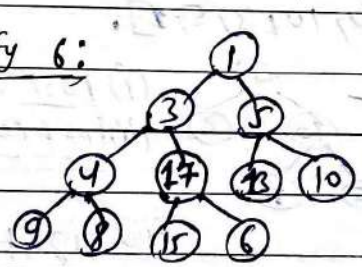
* The task is to build a max-heap from above array.

Total Nodes = 11. ∴ Last Non-leaf node index = (9/2) = 4

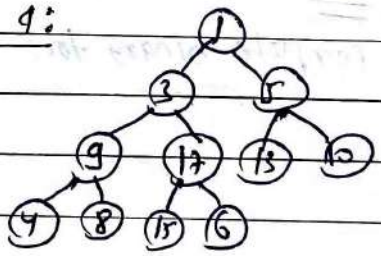
∴, Last non-leaf node = arr[4] = 6.

To build the heap; heapify only the nodes: [1, 3, 5, 4, 6] in reverse.

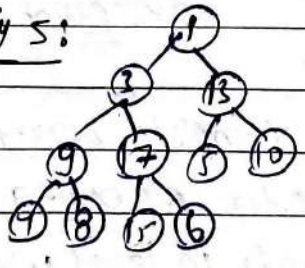
Heapify 6:



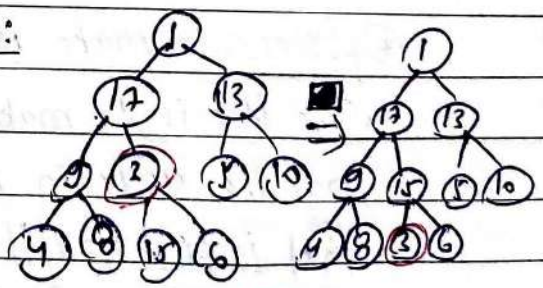
Heapify 4:



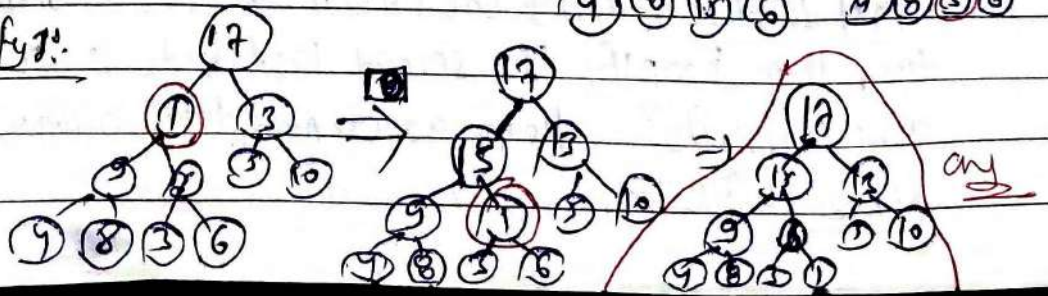
Heapify 5:



Heapify 3:



Heapify 1:



any

(11) $\sum_{h=0}^{\log N} \left\lceil \frac{N}{2^{h+1}} \right\rceil \cdot O(h) = \sum_{h=0}^{\log N} \left\lceil \frac{N}{2^{h+2}} \right\rceil \cdot (ch)$

$= \frac{cN}{2} \left[\sum_{h=0}^{\log n} \left(\frac{h}{2^h} \right) \right] = \frac{cN}{2} \times 2 = O(cN) = O(N)$

Date

Page No. 373.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
void heapify(int arr[], int n, int i){
    int largest = i;
    int l = 2*i+1;
    int r = 2*i+2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i){
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
void buildHeap(int arr[], int n){
    int startIdx = (n-2)/2;
    for(int i = startIdx; i >= 0; i--){ heapify(arr, n, i); }
}
int main(){
    int arr[] = {1,3,5,4,6,13,10,9,8,15,17};
    int n = sizeof(arr)/sizeof(arr[0]);
    buildHeap(arr, n);
    for(int i=0; i<n; i++) cout<<arr[i]<<" ";
    return 0;                    // 17 15 13 9 6 5 10 4 8 3 1
}
```

(4) Heaps in C++ (String built-in class / priority queue):-

- In C++ the priority_queue container can be used to implement Heaps
- priority_queue is built-in container used to implement priority queue
- This container uses max. heap internally to implement a priority queue efficiently. Therefore it can be used in-place of max. heaps.

- This container can also be modified by passing some additional parameters to be used as a Min Heap.

Syntax: For implementing Max Heap.

priority_queue <type_of_data> name_of_heap;

Syntax:- for implementing Min Heap.

priority_queue < type, vector<type>, greater<type>> name_of_heap;

Methods of Priority Queue:

O(1) // • priority_queue::empty():- return whether the queue is empty or not.
O(1) // • priority_queue::size():- return size of the queue.
O(1) // • priority_queue::top():- return reference to the top most element
O(log n) // • priority_queue::push():- The push(g) will add 'g' to end of queue.
O(log n) // • priority_queue::pop():- delete first element of queue.
• priority_queue::swap():- Swap content of two priority queue.

Code:

```
#include<iostream>
#include<queue>
using namespace std;
int main(){
    priority_queue<int> max_heap;          // max Heap
    max_heap.push(10);
    max_heap.push(30);
    max_heap.push(20);
    max_heap.push(5);
    max_heap.push(1);
    (Add element to the max Heap in any order) O(N)

    while(!max_heap.empty())
    {
        cout<<max_heap.top()<<" ";
        max_heap.pop();
    }
    (print element at top of heap & remove element every time from top until become empty)

}
// 30 20 10 5 1
```

```
// min Heap
priority_queue<int, vector<int>, greater<int>> min_heap;
min_heap.push(10);
min_heap.push(30);
min_heap.push(20);
min_heap.push(5);
min_heap.push(1);
while(!min_heap.empty()){
    cout<<min_heap.top()<<" ";
    min_heap.pop();
}
return 0;        // 1 5 10 20 30
}
```

(5) Heap Implementation (Push / Pop) :- (max-Heap)

```
#include<bits/stdc++.h>
using namespace std;
#define SIZE 1001
int heap[SIZE];
int heapSize;
void heap_push(int val){
    if(heapSize>=SIZE){
        cout<<"Overflow\n";
        return;
    }
    heap[heapSize] = val;
    int curr = heapsize;
    while(curr>0 and heap[(curr-1)/2] < heap[curr]){
        int temp=heap[(curr-1)/2];
        heap[(curr-1)/2] = heap[curr];
        heap[curr] = temp;
        curr=(curr-1)/2;
    }
    heapsize+=1
}
```

```
int heap_pop(){
    if (heapSize <=0){
        cout<<"Underflow\n";
        return -1;
    }
    int curr=0;
    int popped = heap[0];
    heap[0]= heap[heapsize-1];
    heapsize -=1;

    while((2*(curr+1)) <heapsize){
        int largest = heap[curr];
        int l = 2*curr+1;
        int r = 2*curr+2;
        if (l<heapsize && heap[l] > heap[largest])
                largest=l;
        if (r< heapsize && heap[r] > heap[largest])
                largest=r;
        if (largest !=curr){
            int temp=heap[curr];
            heap[curr]=heap[largest];
            heap[largest]=temp;
            curr=largest;
        }
        else break;
    }
    return popped;
}
void show_heap(){
    For(int i=0; i<heapsize;++i) cout<<heap[i]<<" ";
    cout<<"\n";
}
```

```
int main(){
    heapSize = 0;
    while (1){
        cout<<"1: Push... 2: POP... 3: Show_Heap... 4: Terminate
        int option, element;
        cin >> option;
        switch (option){
            case 1:
                cout<<"Enter Element In";
                cin>> element;
                heap_push(element);
                break;
            case 2:
                cout<<"popped" << heap_pop()<<"\n";
                break;
            case 3:
                show_heap();
                break;
            default:
                return 0;
        }
    }
}
```

Note :- similarly min heap can be implement with min heap properties.

(6) Heap sort (Ascending Order):-

- it is comparison based sorting technique based on Binary Heap DS.
it is similar to selection sort where we first find the maxm element and place the maximum element at the end. we repeat the same process for remaining element.

Code:-

```
void heapify(arr[], n, i){
    largest = i;
    l = 2*i+1;
    r = 2*i+2;
    if (l<n && arr[largest] < arr[l])  largest = l;
```

```
if (r<n && arr[largest] < arr[r]) largest = r;
if (largest != i){
    swap (arr[i], arr[largest])
    heapify (arr, n, largest)          // O(logn)
}
}
void heapsort (arr[], n){
    for (int i = n/2-1 ; i>=0; i--)    // build heap
        heapify (arr, n, i);           // O(n)
    for (int i=n-1; i>=0; i--){
        swap (arr[0], arr[i]);         // move root to the end
        heapify (arr, i, 0);           // heapify swaped root.
    }                                  // O(n)
}
```

T.C: O(n log n)

(7) Kth Smallest / largest Element in Unsorted Array :-

- Given an array and a number k where k is smaller than size of array. we need to find the kth smallest element in the given array. it is given that elements are distinct

Approach :- create min heap and extract k element from the element, kth extracted element will be the kth smallest element in the array.

code :-

```
[illegible]
    [illegible];
// include <bits/stdc++.h>
using namespace std;
```

```
class minHeap{
    int* harr;
    int capacity;
    int heap_size;
public:
    minHeap(int a[], int size);
    void minHeapify(int i);
    int parent(int i){return (i-1)/2;}
    int left(int i){return (2*i+1);}
    int right(int i){return (2*i+2);}
    int extractMin();
    int getMin{return harr[0];}
};

minHeap::minHeap(int a[], int size){
    heap_size=size;
    harr=a;
    int i=(heap_size-1)/2;
    while(i>=0){
        minHeapify(i);
        i--;
    }
}

int minHeap::extractMin(){
    if(heap_size==0) return INT_MAX;
    int root=harr[0];
    if(heap_size>1){
        harr[0]=harr[heap_size-1];
        meanHeapify(0);
    }
    heap_size--;
    return root;
}
```

```
void MinHeap::MinHeapify(int i){
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if(l< heap_size && harr[l] < harr[i]) smallest = l;
    if(r< heap_size && harr[r] < harr[smallest]) smallest = r;
    if(smallest != i){
        swap(harr[i], harr[smallest]);
        MinHeapify(smallest);
    }
}

int kthSmallest(int arr[], int n, int k){
    MinHeap mh(arr, n);
    for(int i=0; i<k-1; i++) mh.extractMin();
    return mh.getMin();
}

int main(){
    int arr[] = {12, 3, 5, 7, 19};
    int n = sizeof(arr)/sizeof(arr[0]), k=2;
    cout<<"kth smallest element is" << kthSmallest(arr, n, k);
    return 0;
}
```

TC : $O(n + k \log n)$

② using min-heap stl priority queue:-

```
int FindKthLargest(vector<int>& nums, int k){
    priority_queue<int, vector<int>, greater<int>> pq;
    for(int num : nums){
        pq.push(num);
        if(pq.size() > k){
            pq.pop();
        }
    }
    return pq.top();
}
```

② max-heap using priority queue:-

```
int FindKthLargest(vector<int>& nums, int k) {
    priority_queue<int> pq(nums.begin(), nums.end());
    for (int i=0; i<k-1; i++) {
        pq.pop();
    }
    return pq.top();
}
```

⑧ Find Median from data stream:-

The median is the middle value in an ordered integer list, if the size of the list is even, there is no middle value and the median is the mean of the two middle values.

- ex; for arr = [2,3,4], the median is 3.
- for arr = [2,3], the median is (2+3)/2 = 2.5

Implement the MedianFinder class:

- MedianFinder(): initializes the MedianFinder object
- void addNum(int num): adds the integer num from the data stream to DS
- double findMedian(): return the median of element so far in DS.

Process :-

① Create two heaps. one max heap to maintain element of lower half and one min heap to maintain element of higher half at any point of time

② For every newly adding element, insert it into either max heap or min, if there is odd element then one of them will be greater size so lets take max heap as one greater.

case 1 (i) - if, max heap size is zero add element in max heap

case (ii) - else if, both heap size are equal the compare from top of min heap if, number is less than top of min heap then add into max heap. else, pop from minheap add popped element into max heap and add new element to min heap.

Date

Page No. 382.

Case (iii) if both size are not equal i.e maxheap greater

- else, if minheap size is zero but max-heap is not zero:-
if, (inserted) new element is greater than maxheap top element push/add new element to minheap.
else, pop element from maxheap and pushed into minheap and add new element to max heap.
else, if num >= minheap.top() push into minheap.
else num < maxheap.top(), pop from max heap push into min heap and push new element to maxheap.
else, push new element to min-heap.

(3) To calculate median if no. is odd take maxheap top if even take top from both and calculate average.

Code:-

```
class MedianFinder{
  public:
    priority_queue<int> maxheap;
    priority_queue<int, vector<int>, greater<int>> minheap;
    MedianFinder() { };
    void addNum(int num){
        int lsize = maxheap.size();
        int rsize = minheap.size();
Case (i) if (lsize==0) maxheap.push(num);
Case (ii) else if(lsize==rsize){
            if(num<minheap.top()) maxheap.push(num);
            else{
                int temp = minheap.top();
                minheap.pop();
                minheap.push(num);
                maxheap.push(temp);
            }
        }
```

case (iii) else{

```
if (minheap.size()==0){
    if(num>maxheap.top()) minheap.push(num);
    else{
        int temp = maxheap.top();
        maxheap.pop();
        maxheap.push(num);
        minheap.push(temp);
    }
}
else if (num>=minheap.top()) minheap.push(num)
else{
    if (num<maxheap.top()){
        int temp =maxheap.pop();
        maxheap.pop();
        maxheap.push(num);
        minheap.push(temp);
    }
    else   minheap.push(num);
}
}
```

```
double findMedian(){
    int lsize = maxheap.size();
    int rsize =minheap.size();
    if (lsize>rsize) return double(maxheap.top());
    else return (double(maxheap.top())+double(minheap.top()))/2;
}
};
```

TC: O(nlogn) | SC: O(n)

(9) Return k largest element in array.

push the all element one by one in min-heap when min-heap size become greater than k, pop the top element i.e min element at that time. at last we have k size min-heap which is k largest element.

```
void findKlargest(vector<int>& nums, int k){
    priority_queue<int, vector<int>, greater<int>> min_heap;
    for(int num: nums){
        min_heap.push(num);
        if(min_heap.size > k)
            min_heap.pop();
    }
    for(auto num: min_heap)
        cout<<num<<" ";
}
```

TC: O(n) * O(2logk) or, O(nlogk)

(10) Sort a k-Sorted Array / Nearly Sorted

- Given an array of n elements, where each element is at most k away from its target position. sort them.

Ex:- arr[] = {6, 5, 3, 2, 8, 10, 9}, k = 3

k+1

i.e 6 will be at any index (0,1,2)

i.e 5 will be at any index (1,2,3,0)

i.e a element can be before k distance or, after k-distance

.....6, 5, 3, 2, 8, 10, 9.....

2 can be at any index in rang (k+k)

O/P {2, 3, 5, 6, 8, 9, 10}

Approach 1:- use insertion sort

- TC: O(nk), The inner loop (for insertion) will at most k times.

Code:-

```
void insertionSort(int A[], int size){
    for(int i=1; i<size; i++){
        int key = A[i], j = i-1;
        while(j>=0 && A[j]>key){
            A[j+1] = A[j];
            j = j-1;
        }
        A[j+1] = key;
    }
}
```

Approach 2 : using heap

```
void sortK(int arr[], int n, int k){
    int size = (n==k) ? k : k+1;
    priority_queue<int, vector<int>, greater<int>> pq(arr, arr+size);
    int index = 0;
    for(int i=k+1; i<n; i++){
        arr[index++] = pq.top();
        pq.pop();
        pq.push(arr[i]);
    }
    while(pq.empty()==false){
        arr[index++] = pq.top();
        pq.pop();
    }
}
```

TC: O(k) + O(m)*log(k)
here, m=n-k
SC: O(k)

(11) Find k closest numbers in an array.

Given an array and two numbers x and k, find k closest values to x.

Input: arr[] = {10, 2, 14, 4, 7, 6}, X = 5, k = 3

output: 4 6 7 (closest value of 5 is 4 6 7) here.

Process: ① make a max heap of differences with first k elements.

② For every starting from (k+1)th element do-following.

- (i) Find difference of current element with x.
- (ii) If, difference is more than root of heap, ignore current ele.
- (iii) Else, insert the current element after removing the top

③ Finally the heap have k closest elements.

Code:-

```
void printkclosest(int arr[], int n, int x, int k){
    priority_queue<pair<int, int>> pq;
    for(int i=0; i<k; i++)
        pq.push({abs(arr[i]-x), arr[i]});
```

```
for(int i=k; i<n; i++){
    int diff = abs(arr[i]-x);
    if(diff > pq.top().first)
        continue;
    pq.pop();
    pq.push({diff, arr[i]});
}
while(pq.empty()==false){
    cout<< [redacted] pq.top().second <<" ";
    pq.pop();
}
}
```

TC: O(n log k)

(12) Top k Frequent Elements :-

Given an integer array nums and an integer k, return the k most frequent elements.

ex:- nums = [1,1,1,2,2,3], k=2
o/p = [1,2]

ex:- nums = [1], k=1
o/p = [1]

process :-

1. use Hashing technique to count frequency of every number by using map. (O(n) time)
2. use priority queue (heap concept) for frequency. keep the highest frequency 'k' element in heap, (by using min heap we can pop all other element at each step of insertion)

code :-

```
vector<int> topKFrequent(vector<int>& nums, int k){
    unordered_map<int,int> m;
    [redacted]
    for(int i:nums) m[i]++;
    priority_queue<pair<int,int>, vector<pair<int,int>>,
                   greater<pair<int,int>>> p;
```

```
for (auto it : m) {
    p.push ({it.second, it.first});
    if (p.size() > k) p.pop();
}
vector<int> ans;
while (k--) {
    ans.push_back (p.top().second);
    p.pop();
}
return ans;
}
```

TC: O(n) + O(n log k)
SC: O(n) + O(n) + O(n)

Note:- To find top/greater use min heap and to find smaller use max heap.

(13) Frequency Sort:-

print the elements of an array in the decreasing frequency if 2 numbers have same frequency then print the one which came first.

Ex:- arr[] = {2, 5, 2, 8, 5, 6, 8, 8}

O/P : 8, 8, 8, 2, 2, 5, 5, 6.

Process:-

① Take the arr and use unordered_map to have VALUE: FREQUENCY Table

② Then make a HEAP such that high Frequency remains at top and when Frequency is same, just keep in ascending order (smaller at top)

③ Then After Full insertion into Heap. pop one by one and copy it into the array.

TC: O(d*log d) [Dominating factor] (O(n) + 2*d*log(d))

here, d = No. of Distinct element

map insertion | heap / insertion / deletion

SC: O(d)

Code:-

```
class Compare {
    public:
        bool operator() (pair<int,int> a, pair<int,int> b) {
            if (a.first == b.first)
                return a.second > b.second;
            return a.first < b.first;
        }
};

void FrequencySort (vector<int>& arr) {
    int N = arr.size();
    unordered_map<int,int> mp;
    for (int a: arr) mp[a]++;

    priority_queue<pair<int,int>, vector<pair<int,int>>,
                   Compare> pq;
    for (auto m: mp)
        pq.push({m.second, m.first});
    int i=0;
    while (pq.size()>0) {
        int val = pq.top().second;
        int freq = pq.top().first;
        while (freq--) {
            arr[i] = val;
            i++;
        }
        pq.pop();
    }
}
```

(14) K closest points to Origin :-

- Given an array of points where points[i] = [x_i, y_i] represents a point on the x-y plane and an integer k, return the k closest points to the origin.
- The distance b/w two points on x-y plane is Euclidean dist. i.e. $d = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ or, $d^2 = (x_1-x_2)^2+(y_1-y_2)^2$

ex:- points = [[3,3], [5,-1], [2,4]], k=2
o/p: [[3,3], [-2,4]]

process:-
- calculate distance from origin for every point i.e $d^2 = x^2 + y^2$ and make a heap according to d^2 as priority.

code:-
```
vector<vector<int>> kClosest(vector<vector<int>>& points, int k){
    priority_queue<vector<int>> maxHeap.
    For (auto & P: points){
        int x=p[0], y=p[1]
        maxHeap.push({x*x + y*y, x, y});
        if(maxHeap.size()>k){ maxHeap.pop(); }
    }
    vector<vector<int>> ans(k);
    For (int i=0; i<k; ++i){
        vector<int> top = maxHeap.top();
        maxHeap.pop();
        ans[i] = {top[1], top[2]};
    }
    return ans;
}
```

(15) Connect 'n' ropes with minimal cost.

5, 4, 2, 8	5 4 2 8	5 4 2 8
9, 11, 19	6, 11, 19	6, 14, 9
9+11+19 = 39	6+11+19 = 36 (minm)	6+14+9 = 39

code:-
```
int FindMinCost(vector<int>const& prices){
    priority_queue<int, vector<int>, greater<int>> pq(prices.begin(), prices.end());
    int const=0;
```

```
while (pq.size() > 1) {
    int x = pq.top();
    pq.pop();
    int y = pq.top();
    pq.pop();
    int sum = x+y;
    pq.push(sum);
    cost += sum;
}
return cost;
}
```