**############################################################**

**1. Binary Heap Operations**

A **binary heap** is a Binary Tree with the following properties:  
**1)** It’s a *complete tree*(All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

**2)** A Binary Heap is either **Min Heap** or **Max Heap**. In a *Min Binary Heap*, the key at the *root* must be *minimum*among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

You are given an empty Binary Min Heap and some queries and your task is to implement the three methods **insertKey,  deleteKey,** and **extractMin**on the Binary Min Heap and call them as per the query given below:  
**1)***1  x*  (a query of this type means to insert an element in the min-heap with value x )  
**2)** *2  x*  (a query of this type means to remove an element at position x from the min-heap)  
**3)***3*  (a query like this removes the min element from the min-heap and prints it ).

**Example 1:**

**Input:**

Q = 7

**Queries:**

insertKey(4)

insertKey(2)

extractMin()

insertKey(6)

deleteKey(0)

extractMin()

extractMin()

**Output:** 2 6 - 1

**Explanation:** In the first test case for

query

insertKey(4) the heap will have  {4}

insertKey(2) the heap will be {2 4}

extractMin() removes min element from

  heap ie 2 and prints it

  now heap is {4}

insertKey(6) inserts 6 to heap now heap

  is {4 6}

deleteKey(0) delete element at position 0

of the heap,now heap is {6}

extractMin() remove min element from heap

ie 6 and prints it  now the

  heap is empty

extractMin() since the heap is empty thus

no min element exist so -1

  is printed.

**Example 2:**

**Input:**

Q = 5

Queries:

insertKey(8)

insertKey(9)

deleteKey(1)

extractMin()

extractMin()

**Output:** 8 -1

**Your Task:**  
You are required to complete the 3 methods **insertKey()** which take one argument the value to be inserted, **deleteKey()** which takes one argument the position from where the element is to be deleted and **extractMin()** which returns the minimum element in the heap(-1 if the heap is empty)

**Expected Time Complexity:**O(Q\*Log(size of Heap) ).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= **Q** <= 104  
1 <= **x** <= 104

/\*The structure of the class is

struct MinHeap

{

int \*harr;

int capacity, heap\_size;

MinHeap(int cap) {heap\_size = 0; capacity = cap; harr = new int[cap];}

int extractMin();

void deleteKey(int i);

void insertKey(int k);

int parent(int i);

int left(int i);

int right(int i);

};\*/

//Function to extract minimum value in heap and then to store

//next minimum value at first index.

int MinHeap::extractMin()

{

if(heap\_size <= 0)

return -1;

if(heap\_size == 1)

{ heap\_size--;

return \*(harr);

}

int data = \*(harr);

\*harr = \*(harr + heap\_size -1);

heap\_size--;

MinHeapify(0);

return data;

}

//Function to delete a key at ith index.

void MinHeap::deleteKey(int i)

{

if(heap\_size <= 0 || i >= heap\_size)

return;

decreaseKey(i, INT\_MIN);

extractMin();

}

//Function to insert a value in Heap.

void MinHeap::insertKey(int k)

{

if(heap\_size == capacity)

return;

heap\_size++;

\*(harr + heap\_size - 1) = k;

int i = heap\_size - 1;

while(i != 0 && \*(harr + i) < \*(harr + (parent(i))))

{ swap(\*(harr + i), \*(harr + (parent(i))));

i = (parent(i));

}

}

//Function to change value at ith index and store that value at first index.

void MinHeap::decreaseKey(int i, int new\_val)

{

harr[i] = new\_val;

while (i != 0 && harr[parent(i)] > harr[i]) {

swap(harr[i], harr[parent(i)]);

i = parent(i);

}

}

/\* You may call below MinHeapify function in

above codes. Please do not delete this code

if you are not writing your own MinHeapify \*/

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);

}

}

**############################################################**

**2. Heap Sort**

Given an array of size N. The task is to sort the array elements by completing functions **heapify**() and **buildHeap**() which are used to implement Heap Sort.

**Example 1:**

**Input:**

N = 5

arr[] = {4,1,3,9,7}

**Output:**

1 3 4 7 9

**Explanation:**

After sorting elements

using heap sort, elements will be

in order as 1,3,4,7,9.

**Example 2:**

**Input:**

N = 10

arr[] = {10,9,8,7,6,5,4,3,2,1}

**Output:**

1 2 3 4 5 6 7 8 9 10

**Explanation:**

After sorting elements

using heap sort, elements will be

in order as 1, 2,3,4,5,6,7,8,9,10.

**Your Task** **:**  
You don't have to read input or print anything. Your task is to complete the functions **heapify()***,***buildheap()** and **heapSort()**where heapSort() and buildheap() takes the array and it's size as input and heapify() takes the array, it's size and an index i as input. Complete and use these functions to sort the array using heap sort algorithm.

**Expected Time Complexity:** O(N \* Log(N)).  
**Expected Auxiliary Space:** O(1).

**Constraints:**  
1 <= N <= 106  
1 <= arr[i] <= 106

void heapify(int arr[], int n, int i)

{

int l = 2\*i + 1;

int r = 2\*i + 2;

int largest = i;

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);

}

}

//Function to build a Heap from array.

void buildHeap(int arr[], int n)

{

for(int i=n/2 - 1; i >= 0; i--)

heapify(arr, n, i);

}

//Function to sort an array using Heap Sort.

void heapSort(int arr[], int n)

{

buildHeap(arr, n);

for(int i=n-1; i>=0; i--)

{ swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

**############################################################**

**3. Huffman Decoding-1**

Given a string **S**, implement [Huffman Encoding](https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/) and [Decoding](https://www.geeksforgeeks.org/huffman-decoding/).

**Example 1:**

Input : abc

Output : abc

**Example 2:**

Input : geeksforgeeks

Output : geeksforgeeks

**Your task:**  
You don't need to read input or print anything. Your task is to complete the function **decode\_file(),**which takes root of the tree formed while encoding and the encoded string as the input parameters and returns the decoded string.

**Constraints:**  
2<=S<=1000

string decode\_file(struct MinHeapNode\* root, string s)

{

string ans = "";

MinHeapNode \*temp = root;

for(int i=0; i<s.size(); i++)

{

if(s[i] == '0')

temp = temp->left;

else if(s[i] == '1')

temp = temp->right;

if(temp->left == NULL && temp->right == NULL)

{ ans += temp->data;

temp = root;

}

}

ans += '\0';

return ans;

}

**############################################################**

**4. Kth largest element**

Given an array **arr**[] of **N** positive integers and a number K. The task is to find the kth largest element in the array.

**Example 1:**

**Input:**

N = 5, K = 3

arr[] = {3, 5, 4, 2, 9}

**Output:**

4

**Explanation:**

Third largest element

in the array is 4.

**Example 2:**

**Input:**

N = 5, K = 5

arr[] = {4, 3, 7, 6, 5}

**Output:**

3

**Explanation:**

Fifth largest element

in the array is 3.

**Your Task:**  
You are required to complete the method **KthLargest()** which takes 3 arguments and returns the Kth Largest element.

**Constraints:**  
1 <= N <= 106  
1 <= arr[i] <= 106  
1 <= K <= N

**Expected Time Complexity** : O(NlogK)  
**Expected Auxilliary Space**: O(K)

int KthLargest(int arr[], int n, int k)

{

priority\_queue<int, vector<int>, greater<int>> q;

for(int i=0; i<k; i++)

q.push(arr[i]);

for(int i=k; i<n; i++)

{

if(arr[i] >= q.top())

{ q.pop();

q.push(arr[i]);

}

}

return q.top();

}

**############################################################**

**5. Nearly sorted**

Given an array of **n** elements, where each element is at most **k** away from its target position, you need to sort the array optimally.

**Example 1:**

**Input:**

n = 7, k = 3

arr[] = {6,5,3,2,8,10,9}

**Output:** 2 3 5 6 8 9 10

**Explanation:** The sorted array will be

2 3 5 6 8 9 10

**Example 2:**

**Input:**

n = 5, k = 2

arr[] = {3,1,4,2,5}

**Output:** 1 2 3 4 5

**Note: DO NOT** use STL sort() function for this question.

**Your Task:**  
You are required to complete the method **nearlySorted()** which takes 3 arguments and returns the sorted array.

**Expected Time Complexity** : O(nlogk)  
**Expected Auxilliary Space** : O(n)

**Constraints:**  
1 ≤ n ≤ 106  
1 ≤ k < n  
1 ≤ arri≤ 107

vector <int> nearlySorted(int arr[], int n, int k)

{ vector<int> ans;

priority\_queue<int, vector<int>, greater<int>> q;

for(int i=0; i<n; i++)

{ if(i <= k)

q.push(arr[i]);

else

{ q.push(arr[i]);

ans.push\_back(q.top());

q.pop();

}

}

while(!q.empty())

{ ans.push\_back(q.top());

q.pop();

}

return ans;

}

**############################################################**

**6. Minimum Cost of ropes**

There are given **N** ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. The task is to connect the ropes with minimum cost.

**Example 1:**

**Input:**

n = 4

arr[] = {4, 3, 2, 6}

**Output:**

29

**Explanation:**

For example if we are given 4

ropes of lengths 4, 3, 2 and 6. We can

connect the ropes in following ways.

1) First connect ropes of lengths 2 and 3.

Now we have three ropes of lengths 4, 6

and 5.

2) Now connect ropes of lengths 4 and 5.

Now we have two ropes of lengths 6 and 9.

3) Finally connect the two ropes and all

ropes have connected.

Total cost for connecting all ropes is 5

+ 9 + 15 = 29. This is the optimized cost

for connecting ropes. Other ways of

connecting ropes would always have same

or more cost. For example, if we connect

4 and 6 first (we get three strings of 3,

2 and 10), then connect 10 and 3 (we get

two strings of 13 and 2). Finally we

connect 13 and 2. Total cost in this way

is 10 + 13 + 15 = 38.

**Example 2:**

**Input:**

n = 5

arr[] = {4, 2, 7, 6, 9}

**Output:**

62

**Explanation:**

First, connect ropes 4 and 2, which makes

the array {6,7,6,9}. Next, add ropes 6 and

6, which results in {12,7,9}. Then, add 7

and 9, which makes the array {12,16}. And

finally add these two which gives {28}.

Hence, the total cost is 6 + 12 + 16 +

28 = 62.

**Your Task:**  
You don't need to read input or print anything. Your task isto complete the function **minCost()** which takes 2 arguments and returns the minimum cost.

**Expected Time Complexity** : O(nlogn)  
**Expected Auxilliary Space** : O(n)

**Constraints:**  
1 ≤ N ≤ 100000  
1 ≤ arr[i] ≤ 106

long long minCost(long long arr[], long long n)

{

long long cost = 0;

priority\_queue<long long, vector<long long>, greater<long long>> q;

for(long long i=0; i<n; i++)

q.push(arr[i]);

while(q.size() > 1)

{

long long sum = q.top();

q.pop();

sum += q.top();

q.pop();

cost += sum;

q.push(sum);

}

q.pop();

return cost;

}

**############################################################**

**7. Merge k Sorted Arrays**

Given **K** sorted arrays arranged in the form of a matrix of size K\*K. The task is to merge them into one sorted array.  
**Example 1:**

**Input:**

K = 3

arr[][] = {{1,2,3},{4,5,6},{7,8,9}}

**Output:** 1 2 3 4 5 6 7 8 9

**Explanation:**Above test case has 3 sorted

arrays of size 3, 3, 3

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

[7, 8, 9]]

The merged list will be

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

**Example 2:**

**Input:**

K = 4

arr[][]={{1,2,3,4}{2,2,3,4},

{5,5,6,6},{7,8,9,9}}

**Output:**

1 2 2 2 3 3 4 4 5 5 6 6 7 8 9 9

**Explanation:** Above test case has 4 sorted

arrays of size 4, 4, 4, 4

arr[][] = [[1, 2, 2, 2], [3, 3, 4, 4],

[5, 5, 6, 6]  [7, 8, 9, 9 ]]

The merged list will be

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

6, 6, 7, 8, 9, 9 ].

**Your Task:**  
You do not need to read input or print anything. Your task is to complete **mergeKArrays**() function which takes 2 arguments, an arr[K][K] 2D Matrix containing K sorted arrays and an integer K denoting the number of sorted arrays, as input and returns the merged sorted array ( as a pointer to the merged sorted arrays in **cpp,**as an ArrayList in **java,**and list in **python**)

**Expected Time Complexity:** O(K2\*Log(K))  
**Expected Auxiliary Space:** O(K)

**Constraints**:  
1 <= K <= 100

vector<int> mergeKArrays(vector<vector<int>> arr, int k)

{

vector<int> ans;

priority\_queue<int, vector<int>, greater<int>> q;

for(int i=0; i<k; i++)

{ for(int j=0; j<k; j++)

{

q.push(arr[j][i]);

}

ans.push\_back(q.top());

q.pop();

}

while(!q.empty())

{ ans.push\_back(q.top());

q.pop();

}

return ans;

}

**############################################################**

**8. K Most occurring elements**

Given an array **arr**[] of **N**integers in which elements may be repeating several times. Also, a positive number **K** is given and the task is to find sum of total frequencies of K most occurring elements

**Note:**The value of K is guaranteed to be less than or equal to the number of distinct elements in arr.

**Example 1:**

**Input:**

N = 8

arr[] = {3,1,4,4,5,2,6,1}

K = 2

**Output:** 4

**Explanation:** Since, 4 and 1 are 2 most

occurring elements in the array with

their frequencies as 2, 2. So total

frequency is 2 + 2 = 4.

**Example 2:**

**Input:**

N = 8

arr[] = {3,3,3,4,1,1,6,1}

K = 2

**Output:** 6

**Explanation:** Since, 3 and 1 are most

occurring elements in the array with

frequencies 3, 3 respectively. So,

total frequency is 6.

**Your Task:**  
Complete the function **kMostFrequent()**whic returns the frequencies of K most occuring elements.

**Constraints:**  
1 <= N <= 106  
1 <= K <= N  
1 <= arr[i] <= 106

**Expected Time Complexity**: O(N log N)  
**Expected Auxilliary Space:**O(N)

int kMostFrequent(int arr[], int n, int k)

{

unordered\_map<int, int> mp;

for(int i=0; i<n; i++)

mp[arr[i]]++;

priority\_queue<int> q;

for(auto i: mp)

q.push(i.second);

int sum = 0;

for(int i=0; i<k; i++)

{ sum += q.top();

q.pop();

}

return sum;

}

**############################################################**

**9. Find median in a stream**

Given an input stream of **N** integers. The task is to insert these numbers into a new stream and find the median of the stream formed by each insertion of **X** to the new stream.

**Example 1:**

**Input:**

N = 4

X[] = 5,15,1,3

**Output:**

5

10

5

4

**Explanation:**Flow in stream : 5, 15, 1, 3

5 goes to stream --> median 5 (5)

15 goes to stream --> median 10 (5,15)

1 goes to stream --> median 5 (5,15,1)

3 goes to stream --> median 4 (5,15,1 3)

**Example 2:**

**Input:**

N = 3

X[] = 5,10,15

**Output:**

5

7.5

10

**Explanation:**Flow in stream : 5, 10, 15

5 goes to stream --> median 5 (5)

10 goes to stream --> median 7.5 (5,10)

15 goes to stream --> median 10 (5,10,15)

**Your Task:**  
You are required to complete 3 methods **insertHeap()** which takes **x**as argument, **balanceHeaps()**and **getMedian()**and returns the current median.  
**Expected Time Complexity** : O(nlogn)  
**Expected Auxilliary Space** : O(n)

**Constraints:**

1<= N <= 106  
1 <= x <= 106

priority\_queue<int> qmax;

priority\_queue<int, vector<int>, greater<int>> qmin;

void insertHeap(int &x)

{ if(qmax.size() == 0)

qmax.push(x);

else

{

if(x < qmax.top())

qmax.push(x);

else

qmin.push(x);

}

balanceHeaps();

}

//Function to balance heaps.

void balanceHeaps()

{ if(qmin.size() > qmax.size())

{ qmax.push(qmin.top());

qmin.pop();

}

if(qmax.size() - qmin.size() == 2)

{ qmin.push(qmax.top());

qmax.pop();

}

}

//Function to return Median.

double getMedian()

{

if((qmax.size() - qmin.size()) == 1)

return qmax.top();

else

return ((qmax.top() + qmin.top())/2);

}

**############################################################**