**1.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.

**Input:**  
The first line of input contains an integer N denoting the number of elements in the stream. Then the next N lines contains integer x denoting the number to be inserted into the stream.

**Output:**  
For each element added to the stream print the floor of the new median in a new line.

**Constraints:**

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

**Example:**  
**Input:**  
4  
5  
15  
1   
3

**Output:**  
5  
10  
5  
4

CODE:

using namespace std;

int main()

{

int n;

cin>>n;

int arr[n];

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

cin>>arr[i];

priority\_queue<int> l;

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

l.push(arr[0]);

int med=arr[0];

cout<<med<<endl;

for(int i=1;i<n;i++)

{

int x=arr[i];

if(l.size()>r.size())

{

if(x<med)

{

r.push(l.top());

l.pop();

l.push(x);

}

else

r.push(x);

med=(r.top()+l.top())/2;

}

else if(l.size()<r.size())

{

if(x>med)

{

l.push(r.top());

r.pop();

r.push(x);

}

else

l.push(x);

med=(r.top()+l.top())/2;

}

else

{

if(x<med)

{

l.push(x);

med=l.top();

}

else

{

r.push(x);

med=r.top();

}

}

cout<<med<<endl;

}

return 0;

}

**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** **:**  
Complete the functions *heapify() and*buildheap().

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

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

**CODE:**

#include <bits/stdc++.h>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

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

void buildHeap(int arr[], int n);

// main function to do 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);

}

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

for (i=0; i < size; i++)

printf("%d ", arr[i]);

printf("\n");

}

// Driver program to test above functions

int main()

{

int arr[1000000],n,T,i;

scanf("%d",&T);

while(T--){

scanf("%d",&n);

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

scanf("%d",&arr[i]);

heapSort(arr, n);

printArray(arr, n);

}

return 0;

}

// } Driver Code Ends

/\* Main function to do heap sort. This function uses buildHeap()

and heapify()

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);

}

} \*/

// The functions should be written in a way that array become sorted

// in increasing order when above heapSort() is called.

// To heapify a subtree rooted with node i which is an index in arr[].

// n is size of heap. This function is used in above heapSor()

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);

}

// Your Code Here

}

// Rearranges input array so that it becomes a max heap

void buildHeap(int arr[], int n) {

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

heapify(arr,n,i);

// Your Code Here

}

**3. 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 minimumamong 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

CODE:

// Initial Template for C++

#include <bits/stdc++.h>

using namespace std;

typedef long long int ll;

// Structure for Min Heap

struct MinHeap {

int \*harr;

int capacity;

int heap\_size;

// Constructor for Min Heap

MinHeap(int c) {

heap\_size = 0;

capacity = c;

harr = new int[c];

}

~MinHeap() { delete[] harr; }

int parent(int i) { return (i - 1) / 2; }

int left(int i) { return (2 \* i + 1); }

int right(int i) { return (2 \* i + 2); }

void MinHeapify(int); // Implemented in user editor

int extractMin();

void decreaseKey(int i, int new\_val);

void deleteKey(int i);

void insertKey(int k);

};

// Position this line where user code will be pasted.

// Driver code

int main() {

int t;

cin >> t;

while (t--) {

ll a;

cin >> a;

MinHeap h(a);

for (ll i = 0; i < a; i++) {

int c;

int n;

cin >> c;

if (c == 1) {

cin >> n;

h.insertKey(n);

}

if (c == 2) {

cin >> n;

h.deleteKey(n);

}

if (c == 3) {

cout << h.extractMin() << " ";

}

}

cout << endl;

// delete h.harr;

h.harr = NULL;

}

return 0;

}

// } Driver Code Ends

/\*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);

};\*/

// You need to write code for below three functions

/\* Removes min element from min heap and returns it \*/

/\* Removes min element from min heap and returns it \*/

int MinHeap :: extractMin()

{ if(heap\_size == 0)

return -1;

int a = harr[0];

swap(harr[0],harr[heap\_size-1]);

heap\_size--;

MinHeapify(0);

return a;

}

/\* Removes element from position x in the min heap \*/

void MinHeap :: deleteKey(int i)

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

return;

decreaseKey(i,INT\_MIN);

extractMin();

}

/\* Inserts an element at position x into the min heap\*/

void MinHeap ::insertKey(int k)

{

if(heap\_size == capacity)

return;

harr[heap\_size] = INT\_MAX;

heap\_size++;

decreaseKey(heap\_size-1,k);

}

// Decrease Key operation, helps in deleting key from heap

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);

}

}

**4. Rearrange characters**

Given a string S with repeated characters (only lowercase). The task is to rearrange characters in a string such that no two adjacent characters are same.

**Note :** It may be assumed that the string has only lowercase English alphabets.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case contains a single line containing a string of lowercase english alphabets.

**Output:**  
For each test case in a new line print "1" (without quotes) if the generated string doesn't contains any same adjacent characters, else if no such string is possible to be made print "0" (without quotes).

**Constraints:**  
1 <= T <= 100  
1 <= length of string <= 104

**Example:**  
**Input:**  
3  
geeksforgeeks  
bbbabaaacd  
bbbbb

**Output:**  
1  
1  
0

CODE:

using namespace std;

int main()

{

int t;

cin>>t;

while(t--)

{

string s;

cin>>s;

map<char,int> mp;

for(int i=0;s[i]!='\0';i++)

{

mp[s[i]]++;

}

map<char,int> ::iterator it=mp.begin();

vector<int> v;

int sum=0;

while(it!=mp.end())

{

v.push\_back(it->second);

sum+=it->second;

it++;

}

sort(v.begin(),v.end());

//debug

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

// {

// cout<<v[i]<<" ";

// }

// cout<<endl;

if(v.size()<=1)

{

cout<<0<<endl;

continue;

}

else

{

sum=sum-2\*v[v.size()-1];

if(sum>=-1)

cout<<1<<endl;

else

cout<<0<<endl;

}

}

return 0;

}

**5. Kth largest element in a stream**

Given an input stream of **n** integers, find the kth largest element for each element in the stream.

**Input:**  
The first line of input contains an integer **T** denoting the number of test cases. Then T test cases follow. Each test case contains two lines. The first line of each test case contains two space separated integers **k** and **n** . Then in the next line are **n** space separated values of the array.

**Output:**  
For each test case, in a new line, print the space separated values denoting the **kth** largest element at each insertion, if the**kth** largest element at a particular insertion in the stream doesn't exist print **-1**.

**Constraints:**  
1 <= T <= 100  
1 <= K <= n  
1 <= n <= 106  
1 <= stream[] <= 105

**Example:  
Input:**  
2  
4 6  
1 2 3 4 5 6  
1 2  
3 4

**Output:**  
-1 -1 -1 1 2 3  
3 4

CODE:

using namespace std;

int main()

{

int t;

cin>>t;

while(t--)

{

int k,n;

cin>>k>>n;

if(k==1)

{

int maxn=-1;

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

{

int x;

cin>>x;

maxn=max(maxn,x);

cout<<maxn<<" ";

}

cout<<endl;

}

else

{

int arr[n];

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

cin>>arr[i];

priority\_queue<int> l;

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

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

{

if(i+1<k)

{

cout<<"-1"<<" ";

r.push(arr[i]);

}

else

{

int x=arr[i];

if(x>r.top())

{

l.push(r.top());

r.pop();

r.push(x);

}

else

{

l.push(x);

}

cout<<l.top()<<" ";

}

}

cout<<endl;

}

}

return 0;

}

**6. Merge K sorted linked lists**

Given **K** sorted linked lists of different sizes. The task is to merge them in such a way that after merging they will be a single sorted linked list.

**Example 1:**

**Input:**

K = 4

value = {{1,2,3},{4 5},{5 6},{7,8}}

**Output:** 1 2 3 4 5 5 6 7 8

**Explanation:**

The test case has 4 sorted linked

list of size 3, 2, 2, 2

1st    list     1 -> 2-> 3

2nd   list      4->5

3rd    list      5->6

4th    list      7->8

The merged list will be

1->2->3->4->5->5->6->7->8.

**Example 2:**

**Input:**

K = 3

value = {{1,3},{4,5,6},{8}}

**Output:** 1 3 4 5 6 8

**Explanation:**

The test case has 3 sorted linked

list of size 2, 3, 1.

1st list 1 -> 3

2nd list 4 -> 5 -> 6

3rd list 8

The merged list will be

1->3->4->5->6->8.

**Your Task:**  
The task is to complete the function **mergeKList**() which merges the K given lists into a sorted one. The **printing**is done **automatically**by the **driver code**.

**Expected Time Complexity:** O(nk Logk)  
**Expected Auxiliary Space:** O(k)  
**Note: n**is the maximum size of all the **k** link list

**Constraints**  
1 <= K <= 103

**CODE:**

// C++ program to merge k sorted arrays of size n each

#include <bits/stdc++.h>

using namespace std;

// A Linked List node

struct Node

{

int data;

Node\* next;

Node(int x){

data = x;

next = NULL;

}

};

Node\* mergeKLists(Node\* arr[], int N);

/\* Function to print nodes in a given linked list \*/

void printList(Node\* node)

{

while (node != NULL)

{

printf("%d ", node->data);

node = node->next;

}

cout<<endl;

}

// Driver program to test above functions

int main()

{

int t;

cin>>t;

while(t--)

{

int N;

cin>>N;

struct Node \*arr[N];

for(int j=0;j<N;j++)

{

int n;

cin>>n;

int x;

cin>>x;

arr[j]=new Node(x);

Node \*curr = arr[j];

n--;

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

{

cin>>x;

Node \*temp = new Node(x);

curr->next =temp;

curr=temp;

}

}

Node \*res = mergeKLists(arr,N);

printList(res);

}

return 0;

}

// } Driver Code Ends

/\*Linked list Node structure

struct Node

{

int data;

Node\* next;

Node(int x){

data = x;

next = NULL;

}

};

\*/

/\* arr[] is an array of pointers to heads of linked lists

and N is size of arr[] \*/

struct comparator

{

bool operator()(Node\* a,Node\* b)

{

if(a->data>b->data)

return true;

return false;

}

};

Node \* mergeKLists(Node \*arr[], int n)

{

priority\_queue<Node\*,vector<Node\*>,comparator> pq;

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

{

pq.push(arr[i]);

}

Node\* head=NULL;

Node\* last=NULL;

while(!pq.empty())

{

Node\* top=pq.top();

pq.pop();

if(top->next!=NULL)

pq.push(top->next);

if(head==NULL)

{

head=top;

last=top;

}

else

{

last->next=top;

last=top;

}

}

return head;

// Your code here

}