K’th largest element in a stream

Given an infinite stream of integers, find the **K**th largest element at any point of time.

Note: Here we have a stream instead of a whole array and we are allowed to store only K elements.

Examples:

Input: *stream[] = {10, 20, 11, 70, 50, 40, 100, 5, . . .}, K = 3*
Output: *{_, _, 10, 11, 20, 40, 50, 50, . . .}*
Input: *stream[] = {2, 5, 1, 7, 9, . . .}, K = 2*
Output: *{_, 2, 2, 5, 7, . . .}*

Naive Approach: To solve the problem follow the below idea:

Keep an array of size K. The idea is to keep the array sorted so that the Kth largest element can be found in O(1) time (we just need to return the first element of the array, if the array is sorted in increasing order

How to process a new element of the stream?

For every new element in the stream, check if the new element is smaller than the current Kth largest element. If yes, then ignore it. If no, then remove the smallest element from the array and insert the new element in sorted order. The time complexity of processing a new element is O(K)

Kth largest element in a stream using a **self-balancing binary search tree**:

To solve the problem follow the below idea:

Create a self-balancing binary search tree and for every new element in the stream, check if the new element is smaller than the current k’th largest element. If yes, then ignore it. If no, then remove the smallest element from the tree and insert a new element.

The Kth largest element can be found in O(log K) time.

Kth largest element in a stream using a **Min-Heap**:

To solve the problem follow the below idea:

*An **Efficient Solution** is to use a Min Heap of size K to store K largest elements of the stream. The Kth largest element is always at the root and can be found in O(1) time*

How to process a new element of the stream?

Compare the new element with the root of the heap. If a new element is smaller, then ignore it. Otherwise, replace the root with a new element and call heapify for the root of the modified heap

Below is the implementation of the above approach:

- CPP
- Java
- Python
- C#
- Javascript

```cpp
// A C++ program to find k'th
// smallest element in a stream
#include <bits/stdc++.h>
using namespace std;

// Prototype of a utility function
// to swap two integers
void swap(int* x, int* y);

// A class for Min Heap
class MinHeap {
    int* harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int a[], int size); // Constructor
    void buildHeap();
    void MinHeapify(
        int i); // To minheapify subtree rooted with index 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(); // extracts root (minimum) element
    int getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new
    // root
    void replaceMin(int x)
    {
        harr[0] = x;
        MinHeapify(0);
    }
};

MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
}

void MinHeap::buildHeap()
{
    int i = (heap_size - 1) / 2;
    while (i >= 0) {
        MinHeapify(i);
        i--;
```

```
    }
}

// Method to remove minimum element
// (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the minimum value.
    int root = harr[0];

    // If there are more than 1 items,
    // move the last item to
    // root and call heapify.
    if (heap_size > 1) {
        harr[0] = harr[heap_size - 1];
        MinHeapify(0);
    }
    heap_size--;

    return root;
}

// A recursive method to heapify a subtree with root at
// given index This method assumes that the subtrees are
```

```
// already heapified
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);
    }
}

// A utility function to swap two elements
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th largest element from input stream
void kthLargest(int k, vector<int>& A)
{
```

```
// count is total no. of elements in stream seen so far
int count = 0, x; // x is for new element

// Create a min heap of size k
int* arr = new int[k];
MinHeap mh(arr, k);

for (auto& x : A) {

    // Nothing much to do for first k-1 elements
    if (count < k - 1) {
        arr[count] = x;
        count++;

        cout << "Kth largest element is -1 " << endl;
    }

    else {
        // If this is k'th element, then store it
        // and build the heap created above
        if (count == k - 1) {
            arr[count] = x;
            mh.buildHeap();
        }

        else {
            // If next element is greater than
```

```
                // k'th largest, then replace the root
                if (x > mh.getMin())
                    mh.replaceMin(x); // replaceMin calls
                                      // heapify()
            }

            // Root of heap is k'th largest element
            cout << "Kth largest element is "
                 << mh.getMin() << endl;
            count++;
        }
    }
}

// Driver code
int main()
{
    vector<int> arr = { 1, 2, 3, 4, 5, 6 };
    int K = 3;

    // Function call
    kthLargest(K, arr);
    return 0;
}
```


Output

```
Kth largest element is -1
Kth largest element is -1
```

```
Kth largest element is 1
Kth largest element is 2
Kth largest element is 3
Kth largest element is 4
```

Time Complexity: O(N * log K)
Auxiliary Space: O(K)

Below is the implementation of the above approach using **priority-queue**:

- C++
- Java
- Python3
- C#
- Javascript

```cpp
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;

vector<int> kthLargest(int k, int arr[], int n)
{
    vector<int> ans(n);

    // Creating a min-heap using priority queue
    priority_queue<int, vector<int>, greater<int> > pq;

    // Iterating through each element
    for (int i = 0; i < n; i++) {
        // If size of priority
        // queue is less than k
```

```
        if (pq.size() < k)
            pq.push(arr[i]);
        else {
            if (arr[i] > pq.top()) {
                pq.pop();
                pq.push(arr[i]);
            }
        }

        // If size is less than k
        if (pq.size() < k)
            ans[i] = -1;
        else
            ans[i] = pq.top();
    }

    return ans;
}

// Driver Code
int main()
{
    int n = 6;
    int arr[n] = { 1, 2, 3, 4, 5, 6 };
    int k = 4;

    // Function call
```

```
    vector<int> v = kthLargest(k, arr, n);
    for (auto it : v)
        cout << it << " ";
    return 0;
}
```


Output

```
-1 -1 -1 1 2 3
```

Time Complexity: O(N * log K)
Auxiliary Space: O(K)