|  |
| --- |
| **/\* 1) Given an infinite stream of integers, return the element representing the k th largest element in the stream. (Hint: Use Min-heap) \*/**  // Min heap  #include <bits/stdc++.h>  using namespace std;  void Adjust(vector<int>& A, int i, int n)  {  int j = 2 \* i + 1; // Left child index , Adjusted for 0-based index  int item = A[i]; // Store the current node value  while (j < n)  {  // If the right child exists and is greater than the left child  if ((j + 1 < n) && (A[j] > A[j + 1])) // Compare with right child  j = j + 1;  if (item <= A[j])  break;  A[(j - 1) / 2] = A[j];  j = 2 \* j + 1;  }  A[(j - 1) / 2] = item;  }  void Heapify(vector<int>& A, int n)  {  for (int i = (n - 1) / 2; i >= 0; i--)  Adjust(A, i, n);  }  // Function to insert into a min-heap of fixed size k  void insertKthLargest(vector<int>& minHeap, int num, int k)  {  // If the heap has fewer than k elements, just add the new element  if (minHeap.size() < k)  {  minHeap.push\_back(num);  Heapify(minHeap, minHeap.size());  }  // If the heap already has k elements, check if the new number should replace the root  else if (num > minHeap[0])  {  minHeap[0] = num; // Replace root with new element  Adjust(minHeap, 0, k); // Re-adjust heap to maintain min-heap property  }  }  int main()  {  int k = 3; // Example: Finding the 3rd largest element  vector<int> minHeap; // Min-heap to store k largest elements  vector<int> stream = {10, 20, 11, 70, 50, 40, 90}; // Example stream  cout << "Processing stream:" << endl;  for (int num : stream) {  insertKthLargest(minHeap, num, k);  if (minHeap.size() == k)  cout << "After inserting " << num << ", " << k << "-th largest so far is " << minHeap[0] << endl;  else  cout << "After inserting " << num << ", less than " << k << " elements in stream." << endl;  }  return 0;  }  **Output:**    **/\* 2) Suppose a hospital's emergency room is filled with individuals of various ages. Sort the patients efficiently so that the oldest patients receive care first. (Hint: Use Max-heap) \*/**  // Max heap  #include <bits/stdc++.h>  #include <chrono>  using namespace std;  void Adjust(vector<int>& A, int i, int n)  {  int j = 2 \* i ; // Left child index , Adjusted for 0-based index  int item = A[i]; // Store the current node value  while (j <= n)  {  // If the right child exists and is greater than the left child  if (j < n && A[j] > A[j + 1])  j++; // Move to right child  // If the item is greater than or equal to the largest child, we break  if (item <= A[j])  break;    // Move the child up to the parent  A[j/2] = A[j];  // Move down to the child  j = 2 \* j; // Update the child index  }  A[j/2] = item; // Place the item at its correct position  }  void Heapify(vector<int>& A, int n)  {  for (int i = (n / 2) ; i >= 0; i--)  Adjust(A, i, n);  }  void HeapSort(vector<int>& A, int n)  {  Heapify(A, n); // Build max-heap  for (int i = n; i >= 1; i--)  {  // Swap the root of the heap (max element) with the last element  int t = A[i];  A[i] = A[0];  A[0] = t;  Adjust(A, 0, i - 1); // Adjust the heap  }  }  int main()  {  // Example patient ages  vector<int> A = {2, 9, 7, 6, 5, 8, 5, 8, 10, 23};  cout << "Original ages: ";  for (int age : A)  cout << age << " "; // Display original ages  cout << endl;  HeapSort(A, A.size() - 1); // Sort the ages  cout << "Sorted by priority (oldest to youngest): ";  for (int age : A)  cout << age << " "; // Display sorted ages  cout << endl;  return 0;  }  **Output:**    **/\* 3) You are given k sorted arrays, each containing n integers. Write a function that efficiently merges these k sorted arrays into a single sorted array. (Hint: Use Min-heap) \*/**  // Min-heap implementation to merge k sorted arrays  #include <bits/stdc++.h>  #include <chrono>  using namespace std;  // Adjust the heap to maintain the min-heap property  void Adjust(vector<int>& A, int i, int n)  {  int j = 2 \* i; // Start with the left child  int item = A[i];  while (j <= n)  {  if (j <= n && A[j] > A[j + 1]) // Compare with right child  j = j + 1;  if (item <= A[j])  break;  A[j/2] = A[j];  // Move down to the child  j = 2 \* j; // Update the child index  }  A[j/2] = item; // Place the item at its correct position  }  // Function to heapify the array  void Heapify(vector<int>& A, int n)  {  for (int i = (n / 2) ; i >= 0; i--)  Adjust(A, i, n);  }  // Function to merge k sorted arrays  vector<int> mergeKSortedArrays(vector<vector<int>>& arrays)  {  vector<int> heap;  // Push the first element of each array into the heap  for (const auto& arr : arrays)  {  for (int num : arr)  heap.push\_back(num);  }  int n = heap.size();  Heapify(heap, n - 1); // Create a min-heap from the elements  vector<int> result;  // Extract elements from the min-heap  for (int i = n - 1; i >= 0; i--)  {  result.push\_back(heap[0]); // Get the minimum element  heap[0] = heap[i]; // Move the last element to the root  Adjust(heap, 0, i - 1); // Adjust the heap  }  return result; // Return the merged array  }  int main()  {  vector<vector<int>> arrays = {  {1, 4, 17, 10},  {2, 5, 8},  {7 ,3, 6, 9 }  };  vector<int> mergedArray = mergeKSortedArrays(arrays);  cout << "Merged sorted array: ";  for (int num : mergedArray)  cout << num << " ";  cout << endl;  return 0;  }  **Output:** |