📈 Finding the Kth Largest Element using a Min Heap

✅ Problem Statement:

Find the **kth largest element** from an unsorted array using **efficient heap-based logic**, without fully sorting the array.

🧠 Core Idea (Logic):

We use a **Min Heap** to keep track of the **k largest elements** in the array.

- Insert elements into the Min Heap.
- If heap size exceeds `k`, remove the **smallest** element (top of Min Heap).
- After processing all elements, heap contains the **k largest values**.
- The **top element of Min Heap** is the **kth largest**, because it's the smallest among the k largest.

🧾 Corrected C++ Code:

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;

int main() {
  vector<int> v = {10, 20, -4, 6, 1, 24, 105, 118};
  int k = 4;  // Find 4th largest element

  // Min Heap (by default priority_queue is max heap, so we invert using greate
r<int>)
```

```
  priority_queue<int, vector<int>, greater<int>> pq;

  for(int i = 0; i < v.size(); i++) {
    pq.push(v[i]);          // Push current element
    if(pq.size() > k)       // If size exceeds k
      pq.pop();             // Remove the smallest (top of min heap)
  }

  cout << pq.top();         // kth largest element
  return 0;
}
```

📊 Dry Run:

Input: `v = {10, 20, -4, 6, 1, 24, 105, 118}`, `k = 4`

We maintain a **Min Heap of k largest elements**:

Step	Element	Heap (Min Heap)	Action
1	10	[10]	Push
2	20	[10, 20]	Push
3	-4	[-4, 20, 10]	Push
4	6	[-4, 6, 10, 20]	Push
5	1	[-4, 1, 10, 20, 6]	Push → Size > k → Pop -4
		[1, 6, 10, 20]	After popping -4
6	24	[1, 6, 10, 20, 24]	Push → Size > k → Pop 1
		[6, 20, 10, 24]	After popping 1
7	105	[6, 20, 10, 24, 105]	Push → Size > k → Pop 6
		[10, 20, 105, 24]	After popping 6
8	118	[10, 20, 105, 24, 118]	Push → Size > k → Pop 10
		[20, 24, 105, 118]	Final state of Min Heap

✅ Final Answer: **20**, which is the **4th largest element**.

🧮 Time and Space Complexity:

⏱️ Time Complexity:

- Each `push` / `pop` operation in a heap: **O(log k)**
- Total `n` elements processed → **O(n log k)**

📦 Space Complexity:

- Heap stores `k` elements at max → **O(k)**

🔁 Use Case:

- When you want to find the **kth largest** element quickly without full sorting.
- Efficient for large datasets when `k` is much smaller than `n`.

⚠️ Note:

- In C++, the default `priority_queue` is a **Max Heap**.

  To create a **Min Heap**, use this syntax:

  ```
  priority_queue<int, vector<int>, greater<int>> pq;
  ```

🔚 Final Summary:

By keeping a **Min Heap of size k**, and always removing the smallest element when size exceeds `k`,

we ensure that the **heap ends up with the k largest values**, and the **top is exactly the kth largest**.
