

Check Max Heap in C++

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

class Solution {
public:
    static bool checkMaxHeap(vector<int>& arr) {
        for (int i = 0; i < arr.size(); i++) {
            int pIndex = i;
            int lIndex = 2 * i + 1;
            int rIndex = 2 * i + 2;

            if (lIndex < arr.size() && arr[pIndex] <
arr[lIndex]) {
                return false;
            }

            if (rIndex < arr.size() && arr[pIndex] <
arr[rIndex]) {
                return false;
            }
        }
        return true;
    }
};

int main() {
    // Example input
    vector<int> arr = {42, 20, 18, 6, 14, 11, 9, 4};

    // Call the static method checkMaxHeap from
    Solution class
    bool result = Solution::checkMaxHeap(arr);

    // Print the result
    cout << boolalpha << result << endl;

    return 0;
}
```

Dry Run for Input: {42, 20, 18, 6, 14, 11, 9, 4}

Index (i)	Parent (arr[i])	Left Child Index (2i+1)	Left Value	Right Child Index (2i+2)	Right Value	Valid?
0	42	1	20	2	18	✓
1	20	3	6	4	14	✓
2	18	5	11	6	9	✓
3	6	7	4	8 (invalid)	—	✓
4 to 7	Leaf nodes	No children	—	—	—	✓

All parent nodes are greater than their children → ✓
Valid Max Heap

📦 Output:

true

true

SortKSortedArray in C++

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

class KthLargest {
public:
    static int kthLargest(int n, vector<int>& input, int k) {
        // Use a priority queue (max heap) to find the kth largest element
        priority_queue<int> pq;

        // Insert all elements into the max heap
        for (int i = 0; i < n; i++) {
            pq.push(input[i]);
        }

        // Remove the top k-1 elements to get the kth largest element
        for (int j = 0; j < k - 1; j++) {
            pq.pop();
        }

        // Return the kth largest element
        return pq.top();
    }
};

int main() {
    // Example input
    vector<int> arr = {2, 4, 1, 9, 6, 8};
    int k = 3;

    // Call the static method kthLargest from KthLargest class
    int result = KthLargest::kthLargest(arr.size(), arr, k);

    // Print the result
    cout << "Kth largest element: " << result << endl;

    return 0;
}
```

Input:

arr = {2, 4, 1, 9, 6, 8}
k = 3

Dry Run Table:

Step	Action	Heap (Max-Heap structure)	Top Element
Init	Empty		
Insert 2	pq.push(2)	[2]	2
Insert 4	pq.push(4)	[4, 2]	4
Insert 1	pq.push(1)	[4, 2, 1]	4
Insert 9	pq.push(9)	[9, 4, 1, 2]	9
Insert 6	pq.push(6)	[9, 6, 1, 2, 4]	9
Insert 8	pq.push(8)	[9, 6, 8, 2, 4, 1]	9
Pop #1	pq.pop()	[8, 6, 1, 2, 4]	8
Pop #2	pq.pop()	[6, 4, 1, 2]	6

→ Final result = 6 (3rd largest)

Output:

Kth largest element: 6

SortKSortedArray in C++

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

void sort(int arr[], int n, int k) {
    // Create a min-heap (priority_queue) to store the
    // first k+1 elements
    priority_queue<int, vector<int>, greater<int>> pq;

    // Insert the first k+1 elements into the min-heap
    for (int i = 0; i <= k && i < n; i++) {
        pq.push(arr[i]);
    }

    // Process the remaining elements
    int index = 0;
    for (int i = k + 1; i < n; i++) {
        // Pop the smallest element from the min-heap
        // and store it in arr
        arr[index++] = pq.top();
        pq.pop();

        // Push the current element into the min-heap
        pq.push(arr[i]);
    }

    // Pop and store the remaining elements from the
    // min-heap
    while (!pq.empty()) {
        arr[index++] = pq.top();
        pq.pop();
    }
}

int main() {
    int arr[] = {2, 4, 1, 9, 6, 8};
    int k = 3;
    int n = sizeof(arr) / sizeof(arr[0]);

    sort(arr, n, k);

    // Print sorted array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Input:

- arr[] = {2, 4, 1, 9, 6, 8}
- k = 3
- n = 6

🧠 Understanding the Flow:

1. Initialize a **min-heap** (using priority_queue with greater<int>).
2. Push the first k + 1 = 4 elements into the heap: [2, 4, 1, 9]
3. Pop the smallest from the heap and replace in arr (heapify and continue).
4. Keep pushing the next element and popping from the heap until all elements are processed.
5. At the end, empty the remaining heap into the array.

🔍 Dry Run Table:

Step	Min-Heap (Top = Min)	Array Update (arr[])
Init	[1, 2, 4, 9]	—
Pop	1 → arr[0] = 1	[1 , _, _, _, _, _]
Push 6 → Heap =	[2, 6, 4, 9]	—
Pop	2 → arr[1] = 2	[1, 2 , _, _, _, _]
Push 8 → Heap =	[4, 6, 9, 8]	—
Pop	4 → arr[2] = 4	[1, 2, 4 , _, _, _]
No more to push	—	—
Pop	6 → arr[3] = 6	[1, 2, 4, 6 , _, _]
Pop	8 → arr[4] = 8	[1, 2, 4, 6, 8 , _]
Pop	9 → arr[5] = 9	[1, 2, 4, 6, 8, 9]

✅ Final Output:

1 2 4 6 8 9

1 2 4 6 8 9