First non-repeating character in C++

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

class FirstNonRepeatingCharacter {
public:
    string FirstNonRepeating(string A) {
        queue<char> q;
        unordered_map<char, int> hm;
        string ans(A.length(), '#');

        for (int i = 0; i < A.length(); i++) {
            char c = A[i];

            q.push(c);
            hm[c]++;

            while (!q.empty() && hm[q.front()] > 1) {
                q.pop();
            }

            if (!q.empty()) {
                ans[i] = q.front();
            }
        }

        return ans;
    }
};

int main() {
    // Hardcoded input string
    string A = "aabc";

    // Create an instance of the FirstNonRepeatingCharacter class
    FirstNonRepeatingCharacter solution;

    // Call the FirstNonRepeating method and store the result
    string result = solution.FirstNonRepeating(A);

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

    return 0;
}
```

Code Summary:

- Use a **queue** to maintain the order of characters.
- Use a **hash map** (unordered_map<char, int>) to count character occurrences.
- At each step:
    - Add current character to the queue.
    - Increment its count.
    - Remove characters from the front of the queue if their count > 1.
    - The front of the queue (if any) is the current **first non-repeating** character.

🎬 Dry Run for A = "aabc"

i	A[i]	Queue	Hash Map	First Non-Repeating	ans
0	'a'	a	a:1	a	a
1	'a'	a a	a:2	# (a is repeated)	a#
2	'b'	a a b → b	a:2, b:1	b	a#b
3	'c'	b c	a:2, b:1, c:1	b	a#bb

🧾 Final Output:

a#bb

✅ Explanation:

- After 'a': only 'a' is in stream → 'a'
- After second 'a': 'a' repeats → '#'
- After 'b': 'b' is first non-repeating → 'b'
- After 'c': 'b' is still non-repeating → 'b'

```
a#bb
```

Generate Binary in C++

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

vector<string> generate(int N) {
    vector<string> ans;
    queue<string> q;
    q.push("1");
    while (N-- > 0) {
        string rem = q.front();
        q.pop();
        ans.push_back(rem);
        q.push(rem + "0");
        q.push(rem + "1");
    }
    return ans;
}

int main() {
    int N = 5;
    vector<string> binaryNumbers = generate(N);
    for (string num : binaryNumbers) {
        cout << num << endl;
    }
    return 0;
}
```

Goal:

Generate the first N binary numbers (as strings) from 1 to the binary representation of N.

⚙ Algorithm Overview:

- Use a **queue** to build binary numbers level-by-level (like a binary tree).
- Start with "1", then append "0" and "1" to each popped string.
- Do this N times.

🔁 Dry Run for N = 5

Iteration	Queue (Before Pop)	Popped (rem)	Added to Result	Queue (After Push)
1	["1"]	"1"	"1"	["10", "11"]
2	["10", "11"]	"10"	"10"	["11", "100", "101"]
3	["11", "100", "101"]	"11"	"11"	["100", "101", "110", "111"]
4	["100", "101", "110", "111"]	"100"	"100"	["101", "110", "111", "1000", "1001"]
5	["101", "110", "111", "1000", "1001"]	"101"	"101"	["110", "111", "1000", "1001", "1010", "1011"]

📤 Final Output:

```
1
10
11
100
101
```

```
1
10
11
100
101
```

Kth number in C++

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

string kth(int k) {
    queue<string> q;
    q.push("1");
    q.push("2");

    string ans;
    for (int i = 0; i < k; i++) {
        string temp = q.front();
        q.pop();
        ans = temp;
        q.push(temp + "1");
        q.push(temp + "2");
    }

    return ans;
}

int main() {
    int k = 5;
    cout << kth(k) << endl;
    return 0;
}
```

Initial Setup:

```
queue<string> q;
q.push("1");
q.push("2");
```

Initial queue: ["1", "2"]

Dry Run Table:

Iteration (i)	Queue Before	temp (popped)	ans	Queue After Push
0	["1", "2"]	"1"	"1"	["2", "11", "12"]
1	["2", "11", "12"]	"2"	"2"	["11", "12", "21", "22"]
2	["11", "12", "21", "22"]	"11"	"11"	["12", "21", "22", "111", "112"]
3	["12", "21", "22", "111", "112"]	"12"	"12"	["21", "22", "111", "112", "121", "122"]
4	["21", "22", "111", "112", "121", "122"]	"21"	**"21"**	["22", "111", "112", "121", "122", "211", "212"]

📤 **Final Output:**

```
cout << kth(5);
```

Since index starts at 0, on the **5th iteration** (i = 4), we return:

```
21
```

💡 **Output:**

```
21
```

```
21
```

Reverse k elements in C++

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

queue<int> modifyQueue(queue<int> q, int k) {
    stack<int> st;

    // Push the first k elements into a stack
    for (int i = 0; i < k; i++) {
        st.push(q.front());
        q.pop();
    }

    // Pop elements from the stack and enqueue them back into the queue
    while (!st.empty()) {
        q.push(st.top());
        st.pop();
    }

    // Rotate the remaining elements in the queue
    int size = q.size();
    for (int i = 0; i < size - k; i++) {
        q.push(q.front());
        q.pop();
    }

    return q;
}

int main() {
    // Create a queue and add some elements
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);
    q.push(5);

    // Define the value of k
    int k = 3;

    // Call the modifyQueue function and store the result
    queue<int> result = modifyQueue(q, k);

    // Print the result queue
    while (!result.empty()) {
        cout << result.front() << " ";
        result.pop();
    }
    cout << endl;

    return 0;
}
```

Step-by-Step Execution

Step 1: Push first k elements into a stack

Operation	Stack (Top to Bottom)	Queue
push 1	1	[2, 3, 4, 5]
push 2	2, 1	[3, 4, 5]
push 3	3, 2, 1	[4, 5]

Step 2: Pop from stack and enqueue back

Operation	Stack	Queue
pop 3	2, 1	[4, 5, 3]
pop 2	1	[4, 5, 3, 2]
pop 1	empty	[4, 5, 3, 2, 1]

Step 3: Rotate the remaining size - k elements (5 - 3 = 2 times)

Operation	Queue before	Queue after
move 4	[4, 5, 3, 2, 1]	[5, 3, 2, 1, 4]
move 5	[5, 3, 2, 1, 4]	[3, 2, 1, 4, 5]

✓ Final Queue:

[3, 2, 1, 4, 5]

📤 Output:

```
3 2 1 4 5
```

```
3 2 1 4 5
```

Sliding window maximum in C++

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

class SlidingWindowMaximum {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> ans;
        deque<int> deque;

        // Process the first window of size k separately
        for (int i = 0; i < k; i++) {
            while (!deque.empty() && nums[deque.back()] <= nums[i]) {
                deque.pop_back();
            }
            deque.push_back(i);
        }
        ans.push_back(nums[deque.front()]);

        // Process the rest of the elements
        for (int i = k; i < n; i++) {
            if (!deque.empty() && deque.front() == i - k) {
                deque.pop_front();
            }
            while (!deque.empty() && nums[deque.back()] <= nums[i]) {
                deque.pop_back();
            }
            deque.push_back(i);
            ans.push_back(nums[deque.front()]);
        }

        return ans;
    }
};

int main() {
    SlidingWindowMaximum solution;

    // Example 1
    vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    vector<int> result1 = solution.maxSlidingWindow(nums1, k1);
    cout << "Max sliding window for nums1 and k=" << k1 << ": ";
    for (int num : result1) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Dry Run Table:

Index i	**Element nums[i]**	**Deque (indices)**	**Deque (values)**	**Max in window**
0	1	[0]	[1]	-
1	3	[1]	[3]	-
2	-1	[1, 2]	[3, -1]	**3**
3	-3	[1, 2, 3]	[3, -1, -3]	**3**
4	5	[4]	[5]	**5**
5	3	[4, 5]	[5, 3]	**5**
6	6	[6]	[6]	**6**
7	7	[7]	[7]	**7**

🧠 **Explanation:**

- The deque stores **indices** of elements in the current window.
- It's maintained in **decreasing order of values**.
- For each new element:
    - Remove indices from the back if their value is smaller than current.
    - Remove the front index if it's out of the window range.
    - Push the current index to the deque.
    - The front of the deque always has the index of the **max** of current window.

✅ Final Output:

Max sliding window for nums1 and k=3: 3 3 5 5 6 7

```
Max sliding window for nums1 and k=3: 3 3 5 5 6 7
```

Reverse bits in C++

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

class SlidingWindowMinimum {
public:
    vector<int> getMinimums(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> ans;
        if (k > n) return ans;

        deque<int> deque;

        // Process the first window of size k
        for (int i = 0; i < k; i++) {
            while (!deque.empty() && deque.back() > nums[i]) {
                deque.pop_back();
            }
            deque.push_back(nums[i]);
        }
        ans.push_back(deque.front()); // Store the minimum for the first window

        // Process the rest of the elements
        for (int i = k; i < n; i++) {
            if (deque.front() == nums[i - k]) {
                deque.pop_front(); // Remove the element that is no longer in the window
            }
            while (!deque.empty() && deque.back() > nums[i]) {
                deque.pop_back(); // Maintain the deque in descending order
            }
            deque.push_back(nums[i]);
            ans.push_back(deque.front()); // Store the minimum for the current window
        }
        return ans;
    }
};
int main() {
    SlidingWindowMinimum swm;
    // Test case 1
    vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    vector<int> result1 = swm.getMinimums(nums1, k1);
    cout << "Minimums for nums1 and k=" << k1 << ": ";
    for (int num : result1) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Step-by-Step Dry Run (Tracking All Key Values):

i	nums[i]	Deque (indices)	Deque (values)	Action	Window	Min
0	1	[0]	[1]	Initial push	-	-
1	3	[0, 1]	[1, 3]	3 >= 1, keep 0, push 1	-	-
2	-1	[2]	[-1]	Pop 1 and 0 (both > -1), push 2	[1, 3, -1]	-1
3	-3	[3]	[-3]	Pop 2 (nums[2]=-1 > -3), push 3	[3, -1, -3]	-3
4	5	[3, 4]	[-3, 5]	5 > -3, keep 3, push 4	[-1, -3, 5]	-3
5	3	[3, 5]	[-3, 3]	Pop 4 (5 > 3), keep 3, push 5	[-3, 5, 3]	-3
6	6	[5, 6]	[3, 6]	Pop 3 (index out of range), pop 3 (nums[3]=-3 is out), push 6	[5, 3, 6]	3
7	7	[5, 6, 7]	[3, 6, 7]	7 > 6, keep 6, push 7	[3, 6, 7]	3

✅ **Final Output:**

Minimums for nums1 and k=3: -1 -3 -3 -3 3 3

```
Minimums for nums1 and k=3: -1 -3 -3 -3 3 3
```