Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).
Implement the MyQueue class:
void push(int x) Pushes element x to the back of the queue.
int pop() Removes the element from the front of the queue and returns it.
int peek() Returns the element at the front of the queue.
boolean empty() Returns true if the queue is empty, false otherwise

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

class MyQueue {
private:
    stack<int> stack1;
    stack<int> stack2;
    void transfer() {
        while (!stack1.empty()) {
            stack2.push(stack1.top());
            stack1.pop();
        }
    }

public:
    void push(int x) {
        stack1.push(x);
    }
    int pop() {
        if (stack2.empty()) {
            transfer();
        }
        int front = stack2.top();
        stack2.pop();
        return front;
    }
    int peek() {
        if (stack2.empty()) {
```

```
            transfer();
        }
        return stack2.top();
    }
    bool empty() {
        return stack1.empty() && stack2.empty();
    }
};
int main() {
    MyQueue q;
    q.push(1);
    q.push(2);
    cout << "Peek: " << q.peek() << endl; // 1
    cout << "Pop: " << q.pop() << endl;   // 1
    cout << "Peek: " << q.peek() << endl; // 2
    cout << "Empty: " << (q.empty() ? "True" : "False") << endl; // False
    cout << "Pop: " << q.pop() << endl;   // 2
    cout << "Empty: " << (q.empty() ? "True" : "False") << endl; // True
    return 0;
}
```

You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.
Evaluate the expression. Return an integer that represents the value of the expression.

```
#include <iostream>
#include <stack>
#include <vector>
#include <string>
#include <cstdlib>  // For std::stoi

using namespace std;

int evalRPN(vector<string>& tokens) {
    stack<int> stack;

    for (const string& token : tokens) {
        if (token == "+" || token == "-" || token == "*" || token == "/") {
            // Pop the two operands
            int b = stack.top();
            stack.pop();
            int a = stack.top();
            stack.pop();
```

```
            // Perform the operation
            if (token == "+") {
                stack.push(a + b);
            } else if (token == "-") {
                stack.push(a - b);
            } else if (token == "*") {
                stack.push(a * b);
            } else if (token == "/") {
                stack.push(a / b);
            }
        } else {
            // Convert the string to integer and push onto the stack
            stack.push(stoi(token));
        }
    }
    return stack.top();
}

int main() {
    vector<string> tokens = {"2", "1", "+", "3", "*"};
    cout << "Result: " << evalRPN(tokens) << endl;  // Output should be 9

    vector<string> tokens2 = {"4", "13", "5", "/", "+"};
    cout << "Result: " << evalRPN(tokens2) << endl;

    vector<string> tokens3 = {"10", "6", "9", "3", "/", "-", "*"};
    cout << "Result: " << evalRPN(tokens3) << endl;  // Output should be 27

    return 0;
}
```

Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return the next greater number for every element in nums.
The next greater number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return -1 for this number.

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

vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> result(n, -1);  // Initialize the result array with -1
    stack<int> stk;  // Stack to store indices of nums

    // Traverse the array twice to handle circular nature
    for (int i = 0; i < 2 * n; ++i) {
        while (!stk.empty() && nums[stk.top()] < nums[i % n]) {
            // Found the next greater element
```

```
            int idx = stk.top();
            stk.pop();
            result[idx] = nums[i % n];
        }
        stk.push(i % n);  // Push the index of the current element
    }

    return result;
}

int main() {
    vector<int> nums = {1, 2, 1};
    vector<int> result = nextGreaterElements(nums);

    // Output the result
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;  // Expected output: 2 -1 2

    return 0;
}
```

Given a queue, write a recursive function to reverse it.

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

// Function to reverse the queue recursively
void reverseQueue(queue<int>& q) {
    // Base case: If the queue is empty, return
    if (q.empty()) {
        return;
    }

    // Step 1: Dequeue the front element
    int front = q.front();
    q.pop();

    // Step 2: Recursively reverse the remaining queue
    reverseQueue(q);

    // Step 3: Enqueue the front element to the rear
    q.push(front);
}

// Helper function to display the elements of the queue
void printQueue(queue<int>& q) {
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
```

```
    }
    cout << endl;
}

int main() {
    queue<int> q;

    // Enqueue some elements
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);
    q.push(5);

    cout << "Original Queue: ";
    printQueue(q);

    // Reverse the queue
    reverseQueue(q);

    cout << "Reversed Queue: ";
    printQueue(q);

    return 0;
}
```

Given a balanced parentheses string s, return the score of the string.

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

int scoreOfParentheses(string s) {
    stack<int> stk;

    for (char c : s) {
        if (c == '(') {
            stk.push(-1); // -1 is a marker for an opening parenthesis
        } else {
            int currentScore = 0;

            // While the top of the stack is not -1 (marker for '(')
            while (stk.top() != -1) {
                currentScore += stk.top();
                stk.pop();
            }

            stk.pop(); // Pop the -1 marker

            // If currentScore is 0, it means it's a simple "()", so push 1
            // Otherwise, push 2 * currentScore for a nested structure
```

```
            stk.push(currentScore == 0 ? 1 : 2 * currentScore);
        }
    }

    int totalScore = 0;
    // Sum up all scores in the stack (final score is the sum of all scores)
    while (!stk.empty()) {
        totalScore += stk.top();
        stk.pop();
    }

    return totalScore;
}

int main() {
    string s = "(())()";
    cout << "Score of the parentheses string '" << s << "' is: " << scoreOfParentheses(s) << endl;

    s = "((()(())))";
    cout << "Score of the parentheses string '" << s << "' is: " << scoreOfParentheses(s) << endl;

    return 0;
}
```

You are given a 0-indexed string pattern of length n consisting of the characters 'I' meaning increasing and 'D' meaning decreasing.

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

vector<int> generateSequence(string pattern) {
    int n = pattern.size();
    stack<int> stk;
    vector<int> result(n + 1);

    int num = 1;

    for (int i = 0; i <= n; ++i) {
        // Push the current number onto the stack
        stk.push(num++);

        // If we reach the end of the pattern or the current pattern is 'I'
        // pop all elements from the stack to form a valid sequence
        if (i == n || pattern[i] == 'I') {
            while (!stk.empty()) {
                result[i - stk.size()] = stk.top();
                stk.pop();
            }
        }
    }
```

```
    return result;
}

void printSequence(const vector<int>& seq) {
    for (int num : seq) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    string pattern = "IDID";
    vector<int> result = generateSequence(pattern);

    cout << "Generated sequence: ";
    printSequence(result);  // Expected output: 1 3 2 4 5

    return 0;
}
```

You are given an integer array nums of length n and an integer array queries.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>  // For std::gcd

using namespace std;

vector<int> gcdQuery(const vector<int>& nums, const vector<int>& queries) {
    int n = nums.size();
    vector<int> gcdPairs;

    // Step 1: Compute GCD for all pairs (i, j) where 0 <= i < j < n
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            int g = gcd(nums[i], nums[j]);
            gcdPairs.push_back(g);
        }
    }

    // Step 2: Sort the GCD pairs in ascending order
    sort(gcdPairs.begin(), gcdPairs.end());

    // Step 3: Answer the queries
    vector<int> answer;
    for (int q : queries) {
        answer.push_back(gcdPairs[q]);
    }

    return answer;
```

```
}

// Helper function to print the answer array
void printAnswer(const vector<int>& answer) {
   for (int num : answer) {
      cout << num << " ";
   }
   cout << endl;
}

int main() {
   vector<int> nums = {6, 12, 18};
   vector<int> queries = {0, 1, 2, 3};

   vector<int> result = gcdQuery(nums, queries);

   cout << "Answer for the queries: ";
   printAnswer(result);  // Expected: values at indices from sorted GCD pairs array

   return 0;
}
```

Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

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

int firstUniqChar(string s) {
   unordered_map<char, int> charCount;

   // Step 1: Count the frequency of each character
   for (char c : s) {
      charCount[c]++;
   }

   // Step 2: Find the first character with frequency 1
   for (int i = 0; i < s.length(); i++) {
      if (charCount[s[i]] == 1) {
         return i;  // Return the index of the first non-repeating character
      }
   }

   // If no non-repeating character is found
   return -1;
}

int main() {
   string s = "leetcode";
```

```
    int index = firstUniqChar(s);
    cout << "The first non-repeating character index is: " << index << endl;  // Output: 0

    s = "loveleetcode";
    index = firstUniqChar(s);
    cout << "The first non-repeating character index is: " << index << endl;  // Output: 2

    s = "aabb";
    index = firstUniqChar(s);
    cout << "The first non-repeating character index is: " << index << endl;  // Output: -1

    return 0;
}
```

You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

```
#include <iostream>
#include <vector>
#include <stack>
#include <string>
#include <sstream>
#include <unordered_map>
using namespace std;

int evalRPN(vector<string>& tokens) {
    stack<int> stk;

    // Iterate through each token in the expression
    for (const string& token : tokens) {
        if (token == "+" || token == "-" || token == "*" || token == "/") {
            // If the token is an operator, pop two operands
            int b = stk.top();
            stk.pop();
            int a = stk.top();
            stk.pop();

            // Perform the operation
            int result = 0;
            if (token == "+") {
                result = a + b;
            } else if (token == "-") {
                result = a - b;
            } else if (token == "*") {
                result = a * b;
            } else if (token == "/") {
                result = a / b; // Integer division
            }

            // Push the result back onto the stack
            stk.push(result);
        } else {
```

```
            // If the token is a number, push it onto the stack
            stk.push(stoi(token));
        }
    }

    // The final result is the only element left in the stack
    return stk.top();
}

int main() {
    vector<string> tokens = {"2", "1", "+", "3", "*"};
    cout << "Result: " << evalRPN(tokens) << endl;  // Output: 9

    vector<string> tokens2 = {"4", "13", "5", "/", "+"};
    cout << "Result: " << evalRPN(tokens2) << endl;  // Output: 6

    vector<string> tokens3 = {"10", "6", "9", "3", "/", "-", "*"};
    cout << "Result: " << evalRPN(tokens3) << endl;  // Output: 27

    return 0;
}
```