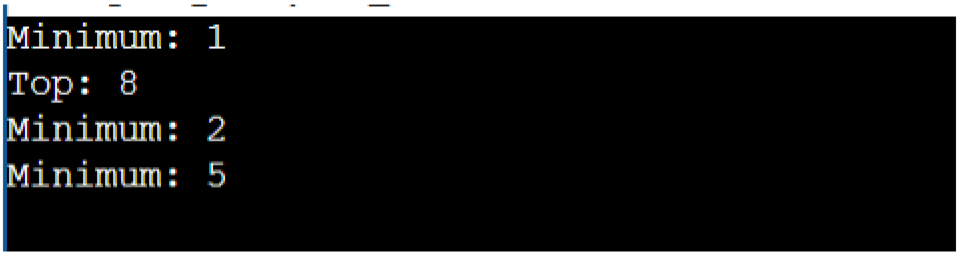
Name: Lakshita **UID**: 22BCS15284

Section: IOT-620 **Date**: 24-12-24

1 Design a stack in c++ that supports push, pop, top, and retrieving the minimum element in constant time.

```
#include <stack>
#include <iostream>
class MinStack {
private:
    std::stack<int> mainStack; // Stack to store all elements
    std::stack<int> minStack;  // Stack to store minimum elements

public:
    // Constructor to initialize the stack object
    MinStack() {
        // Both stacks are initialized empty
    }

    // Pushes the element val onto the stack
    void push(int val) {
        mainStack.push(val);
        // Push to minStack only if it's empty or the new value is less than or equal to the current
minimum
        if (minStack.empty() || val <= minStack.top()) {
            minStack.push(val);
        }
    }

    // Removes the element on the top of the stack
    void pop() {
        if (mainStack.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        // If the top of minStack matches the top of mainStack, pop from minStack as well
        if (mainStack.top() == minStack.top()) {
            minStack.pop();
        }
        mainStack.pop();
    }

    // Gets the top element of the stack
    int top() {
        if (mainStack.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        return mainStack.top();
    }

    // Retrieves the minimum element in the stack
    int getMin() {
        if (minStack.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        return minStack.top();
    }
};

// Example usage
int main() {
    MinStack minStack;
    minStack.push(5);
    minStack.push(2);
    minStack.push(8);
    minStack.push(1);

    std::cout << "Minimum: " << minStack.getMin() << std::endl; // Output: 1
    minStack.pop();
    std::cout << "Top: " << minStack.top() << std::endl;        // Output: 8
    std::cout << "Minimum: " << minStack.getMin() << std::endl; // Output: 2

    minStack.pop();
    minStack.pop();
    std::cout << "Minimum: " << minStack.getMin() << std::endl; // Output: 5

    return 0;
}
```

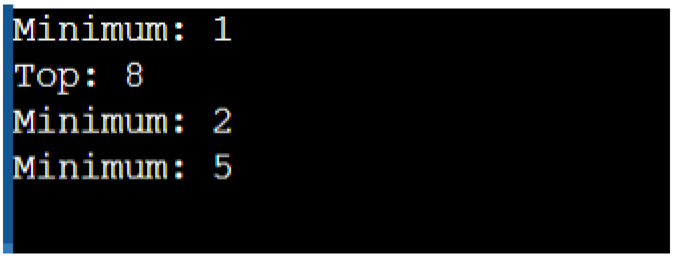
OUTPUT:



```
Minimum: 1
Top: 8
Minimum: 2
Minimum: 5
```

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

```
#include <stack>
#include <iostream>

class MinStack {
private:
    std::stack<int> mainStack; // Stack to store all elements
    std::stack<int> minStack;  // Stack to store minimum elements

public:
    // Constructor to initialize the stack object
    MinStack() {
        // Both stacks are initialized empty
    }

    // Pushes the element val onto the stack
    void push(int val) {
        mainStack.push(val);
        // Push to minStack only if it's empty or the new value is less than or equal to the current
minimum
        if (minStack.empty() || val <= minStack.top()) {
            minStack.push(val);
        }
    }

    // Removes the element on the top of the stack
    void pop() {
        if (mainStack.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        // If the top of minStack matches the top of mainStack, pop from minStack as well
        if (mainStack.top() == minStack.top()) {
            minStack.pop();
        }
        mainStack.pop();
    }

    // Gets the top element of the stack
    int top() {
        if (mainStack.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        return mainStack.top();
    }
    // Retrieves the minimum element in the stack
    int getMin() {
        if (minStack.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        return minStack.top();
    }
};

// Example usage
int main() {
    MinStack minStack;
    minStack.push(5);
    minStack.push(2);
    minStack.push(8);
    minStack.push(1);

    std::cout << "Minimum: " << minStack.getMin() << std::endl; // Output: 1
    minStack.pop();
    std::cout << "Top: " << minStack.top() << std::endl;        // Output: 8
    std::cout << "Minimum: " << minStack.getMin() << std::endl; // Output: 2

    minStack.pop();
    minStack.pop();
    std::cout << "Minimum: " << minStack.getMin() << std::endl; // Output: 5

    return 0;
}
```



```
Minimum: 1
Top: 8
Minimum: 2
Minimum: 5
```

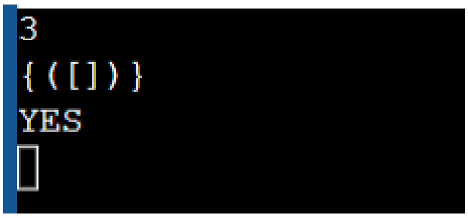
3 Implement a simple text editor. The editor initially contains an empty string, S.Perform Q operations of the following 4 types:

□ append(W) - Append string W to the end of S.
□ delete (k)- Delete the last k characters of S.
□ print (k)- Print the k^th character of S.
□ undo() - Undo the last (not previously undone) operation of type 1 or 2, reverting S to the state it was in prior to that operation.

```
#include <iostream>
#include <stack>
#include <string>

class TextEditor {
private:
    std::string S; // The main string
    std::stack<std::pair<int, std::string>> operations; // Stack to store operations for undo

public:
    // Append string W to the end of S
    void append(const std::string& W) {
        operations.push({1, W}); // Save operation type 1 with appended string
        S += W;
    }

    // Delete the last k characters of S
    void deleteChars(int k) {
        if (k > S.size()) {
            throw std::runtime_error("Cannot delete more characters than available in the string.");
        }
        std::string removed = S.substr(S.size() - k, k);
        operations.push({2, removed}); // Save operation type 2 with removed string
        S.erase(S.size() - k);
    }

    // Print the k^th character of S
    void print(int k) const {
        if (k < 1 || k > S.size()) {
            throw std::runtime_error("Invalid index for print operation.");
        }
        std::cout << S[k - 1] << std::endl;
    }

    // Undo the last operation of type 1 or 2
    void undo() {
        if (operations.empty()) {
            throw std::runtime_error("No operations to undo.");
        }
        auto lastOp = operations.top();
        operations.pop();

        if (lastOp.first == 1) {
            // Undo append operation
            S.erase(S.size() - lastOp.second.size());
        } else if (lastOp.first == 2) {
            // Undo delete operation
            S += lastOp.second;
        }
    }
};

int main() {
    TextEditor editor;
    int Q;
    std::cin >> Q;

    for (int i = 0; i < Q; ++i) {
        int operation;
        std::cin >> operation;

        if (operation == 1) {
            std::string W;
            std::cin >> W;
            editor.append(W);
        } else if (operation == 2) {
            int k;
            std::cin >> k;
            editor.deleteChars(k);
        } else if (operation == 3) {
            int k;
            std::cin >> k;
            editor.print(k);
        } else if (operation == 4) {
            editor.undo();
        } else {
            std::cerr << "Invalid operation code." << std::endl;
        }
    }

    return 0;
}
```

4 A bracket is considered to be any one of the following characters: (,), {, }, [, or].

```
#include <iostream>
#include <stack>
#include <string>

// Function to check if the brackets are balanced
std::string isBalanced(const std::string& s) {
    std::stack<char> bracketStack;

    // Iterate through each character in the string
    for (char c : s) {
        // Push opening brackets onto the stack
        if (c == '(' || c == '{' || c == '[') {
            bracketStack.push(c);
        }
        // Check for closing brackets
        else if (c == ')' || c == '}' || c == ']') {
            if (bracketStack.empty()) {
                return "NO"; // Unmatched closing bracket
            }

            char top = bracketStack.top();
            bracketStack.pop();

            // Ensure the top of the stack matches the current closing bracket
            if ((c == ')' && top != '(') ||
                (c == '}' && top != '{') ||
                (c == ']' && top != '[')) {
                return "NO"; // Mismatched brackets
            }
        }
    }

    // If the stack is not empty, there are unmatched opening brackets
    return bracketStack.empty() ? "YES" : "NO";
}

int main() {
    int n;
    std::cin >> n;

    for (int i = 0; i < n; ++i) {
        std::string s;
        std::cin >> s;
        std::cout << isBalanced(s) << std::endl;
    }

    return 0;
}
```



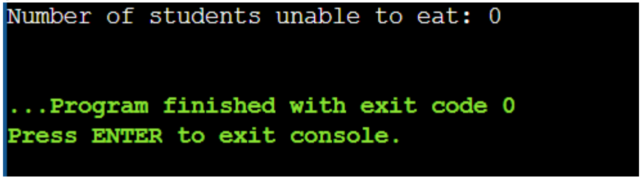
```
3
{([])}
YES
```

5 The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

Soln:

```
#include <iostream>
#include <queue>
```

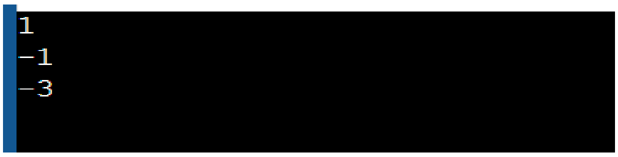
```
#include <vector>
using namespace std;

int countStudents(vector<int>& students, vector<int>& sandwiches) {
    queue<int> studentQueue;
    for (int student : students) {
        studentQueue.push(student);
    }

    int i = 0; // Index for the top sandwich
    int count = 0; // Counter to track attempts without progress

    while (!studentQueue.empty() && count < studentQueue.size()) {
        if (studentQueue.front() == sandwiches[i]) {
            studentQueue.pop(); // The student takes the sandwich and leaves
            i++; // Move to the next sandwich
            count = 0; // Reset the counter since a sandwich was taken
        } else {
            studentQueue.push(studentQueue.front()); // Move the student to the back
            studentQueue.pop(); // Remove the student from the front
            count++; // Increment the counter for failed attempts
        }
    }

    // The remaining students in the queue cannot eat
    return studentQueue.size();
}

int main() {
    vector<int> students = {1, 1, 0, 0};
    vector<int> sandwiches = {0, 1, 0, 1};

    cout << "Number of students unable to eat: " << countStudents(students, sandwiches) << endl;

    return 0;
}
```



```
Number of students unable to eat: 0


...Program finished with exit code 0
Press ENTER to exit console.
```

6 Given an integer array nums, handle multiple queries of the following type:
Calculate the sum of the elements of nums between indices left and right inclusive where left <= right.

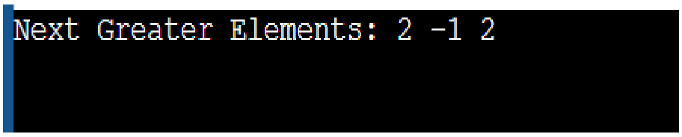
```
#include <iostream>
#include <vector>
using namespace std;

class NumArray {
private:
    vector<int> prefixSum;

public:
    // Constructor to initialize the object and compute the prefix sum array
    NumArray(vector<int>& nums) {
        int n = nums.size();
        prefixSum.resize(n + 1, 0); // Prefix sum array with an extra 0 at the start
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }
    }

    // Method to compute the range sum using the prefix sum array
    int sumRange(int left, int right) {
        return prefixSum[right + 1] - prefixSum[left];
    }
};

int main() {
    // Example usage
    vector<int> nums = {-2, 0, 3, -5, 2, -1};
    NumArray* obj = new NumArray(nums);

    // Queries
    cout << obj->sumRange(0, 2) << endl; // Output: 1
    cout << obj->sumRange(2, 5) << endl; // Output: -1
    cout << obj->sumRange(0, 5) << endl; // Output: -3

    delete obj;
    return 0;
}
```



```
1
-1
-3
```

7 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.

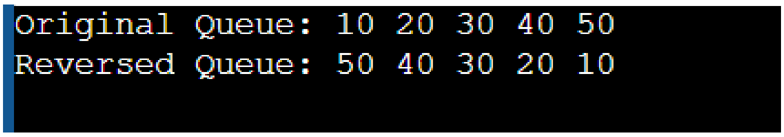
```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

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

    // Traverse the array twice to simulate the circular behavior
    for (int i = 0; i < 2 * n; ++i) {
        int currIndex = i % n; // Current index in the circular array

        // While the stack is not empty and the current element is greater than
        // the element at the index on top of the stack
        while (!s.empty() && nums[currIndex] > nums[s.top()]) {
            result[s.top()] = nums[currIndex]; // Set the next greater element
            s.pop(); // Remove the index from the stack
        }

        // Only push indices from the first traversal to avoid duplicates
        if (i < n) {
            s.push(currIndex);
        }
    }

    return result;
}

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

    cout << "Next Greater Elements: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```



```
Next Greater Elements: 2 -1 2
```

8 Given a queue, write a recursive function to reverse it.
Standard operations allowed :
enqueue(x) : Add an item x to rear of queue.
dequeue() : Remove an item from front of queue.
empty() : Checks if a queue is empty or not.

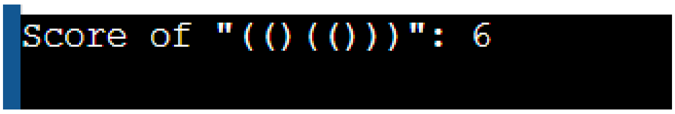
```
#include <iostream>
#include <queue>
using namespace std;

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

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

    // Recursive call to reverse the rest of the queue
    reverseQueue(q);

    // Enqueue the front element to the back of the queue
    q.push(front);
}

int main() {
    // Initialize a queue
    queue<int> q;
    q.push(10);
    q.push(20);
    q.push(30);
    q.push(40);
    q.push(50);

    // Print the original queue
    cout << "Original Queue: ";
    queue<int> temp = q; // Temporary copy of the queue for printing
    while (!temp.empty()) {
        cout << temp.front() << " ";
        temp.pop();
    }
    cout << endl;

    // Reverse the queue
    reverseQueue(q);

    // Print the reversed queue
    cout << "Reversed Queue: ";
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;

    return 0;
}
```



```
Original Queue: 10 20 30 40 50
Reversed Queue: 50 40 30 20 10
```

9 Given a balanced parentheses string s, return the score of the string.
The score of a balanced parentheses string is based on the following rule:
"()" has score 1.
AB has score A + B, where A and B are balanced parentheses strings.
(A) has score 2 * A, where A is a balanced parentheses 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(0); // Start a new score context
        } else {
            // Closing parenthesis: compute the score
            int topScore = stk.top();
            stk.pop();

            // If topScore is 0, it means "()" -> score is 1
            int score = (topScore == 0) ? 1 : 2 * topScore;

            // Add the score to the previous context
            if (!stk.empty()) {
                stk.top() += score;
            } else {
                stk.push(score);
            }
        }
    }

    // The stack contains the final score
    return stk.top();
}

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

    cout << "Score of \"" << s << "\": " << scoreOfParentheses(s) << endl;

    return 0;
}
```



```
Score of "(()(()))": 6
```