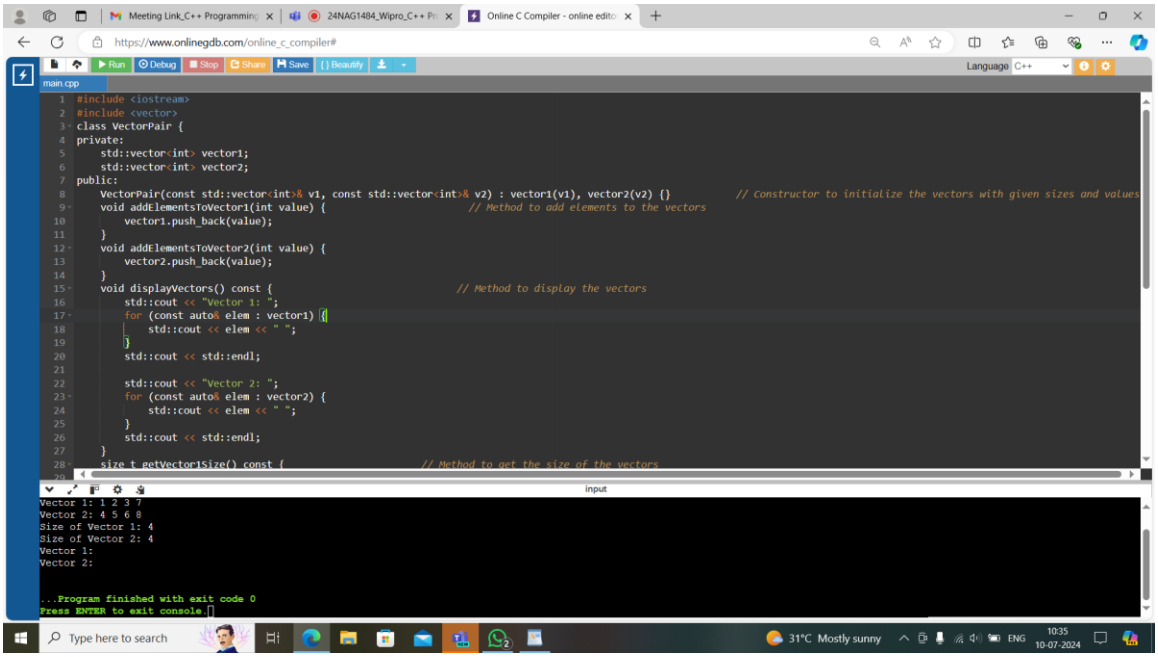
VECTORS :-

```
#include <iostream>

#include <vector>

class VectorPair {

private:

    std::vector<int> vector1;

    std::vector<int> vector2;

public:

    VectorPair(const std::vector<int>& v1, const std::vector<int>& v2) : vector1(v1), vector2(v2) {}
// Constructor to initialize the vectors with given sizes and values

    void addElementsToVector1(int value) {                    // Method to add elements
to the vectors

        vector1.push_back(value);

    }

    void addElementsToVector2(int value) {

        vector2.push_back(value);

    }

    void displayVectors() const {                             // Method to display the
vectors

        std::cout << "Vector 1: ";

        for (const auto& elem : vector1) {

            std::cout << elem << " ";

        }

        std::cout << std::endl;


        std::cout << "Vector 2: ";

        for (const auto& elem : vector2) {
```

```
            std::cout << elem << " ";
        }
        std::cout << std::endl;
    }
    size_t getVector1Size() const {                    // Method to get the size of the vectors
        return vector1.size();
    }

    size_t getVector2Size() const {
        return vector2.size();
    }
    void clearVectors() {                              // Method to clear the vectors
        vector1.clear();
        vector2.clear();
    }
};
int main() {
    std::vector<int> v1 = {1, 2, 3};
    std::vector<int> v2 = {4, 5, 6};
    VectorPair vp(v1, v2);
    vp.displayVectors();
    vp.addElementsToVector1(7);
    vp.addElementsToVector2(8);
    vp.displayVectors();
    std::cout << "Size of Vector 1: " << vp.getVector1Size() << std::endl;
```

```
    std::cout << "Size of Vector 2: " << vp.getVector2Size() << std::endl;
    vp.clearVectors();
    vp.displayVectors();
    return 0;
}
```

OUTPUT :-

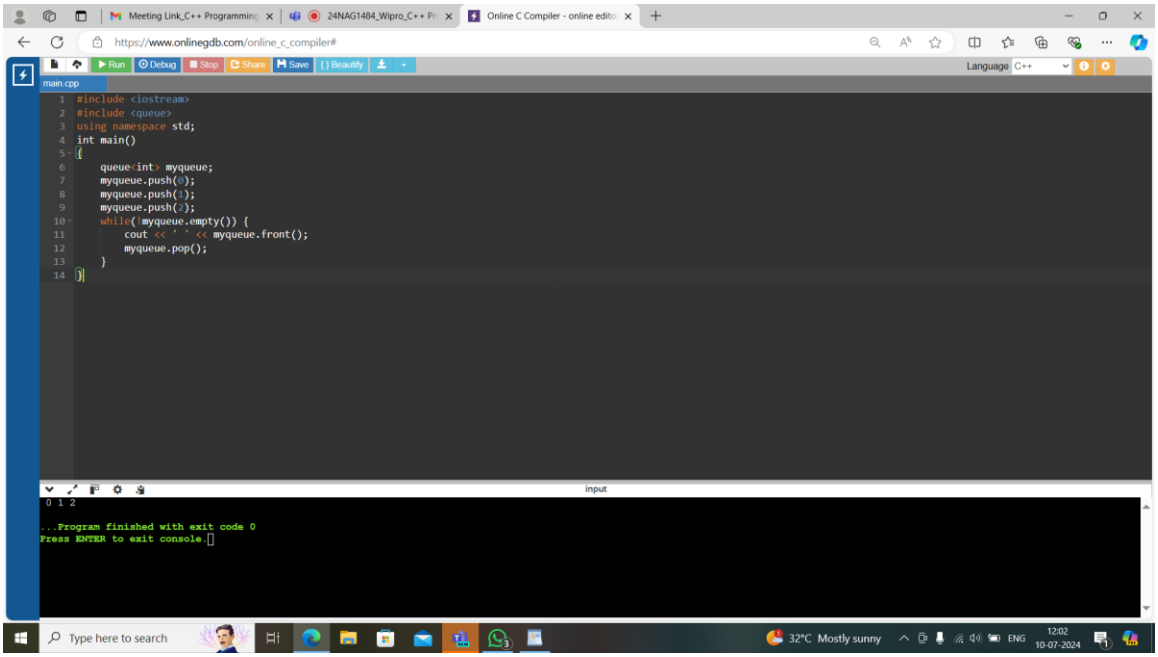


QUEUE::PUSH() :-

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
    queue<int> myqueue;
    myqueue.push(0);
    myqueue.push(1);
```

```
    myqueue.push(2);
    while(!myqueue.empty()) {
        cout << ' ' << myqueue.front();
        myqueue.pop();
    }
}
```

OUTPUT :-

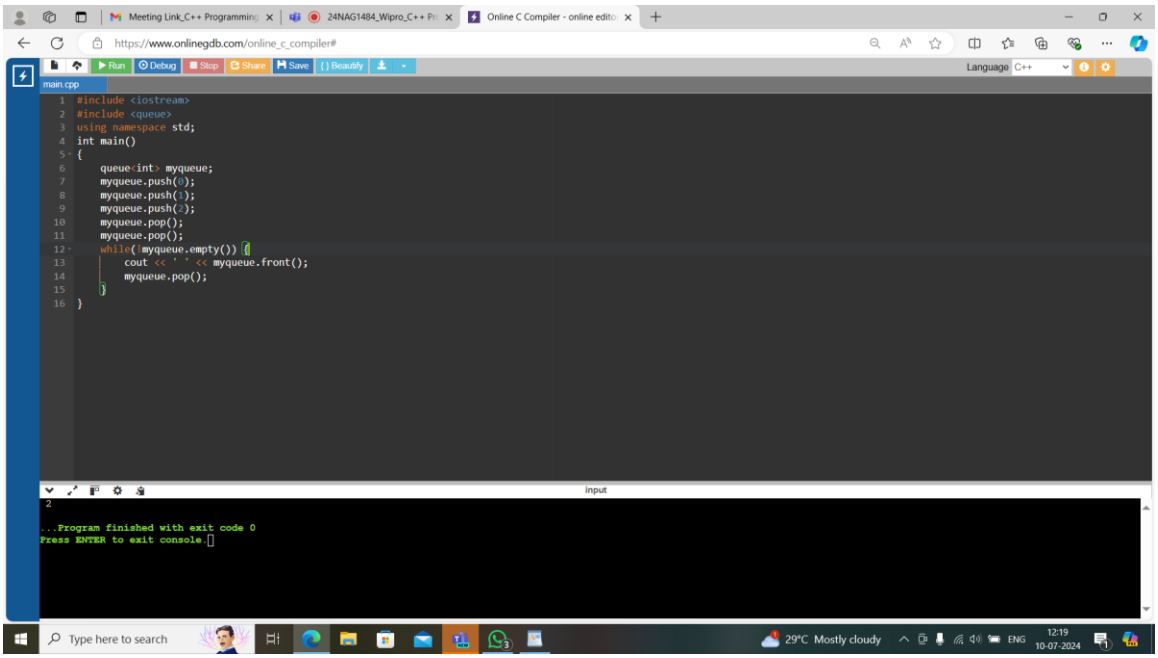


QUEUE:: POP() :-

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
    queue<int> myqueue;
    myqueue.push(0);
    myqueue.push(1);
```

```
    myqueue.push(2);
    myqueue.pop();
    myqueue.pop();
    while(!myqueue.empty()) {
        cout << ' ' << myqueue.front();
        myqueue.pop();
    }
}
```

OUTPUT :-

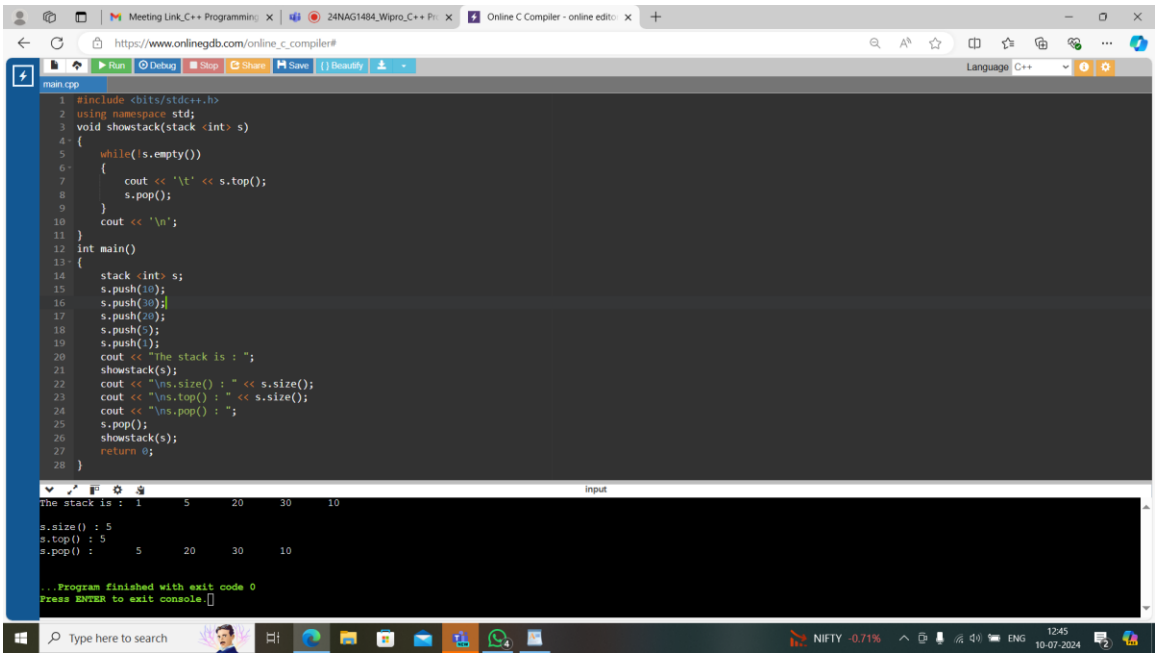


STACK CONTINUED :-

```
#include <bits/stdc++.h>
using namespace std;
void showstack(stack <int> s)
{
    while(!s.empty())
    {
```

```
            cout << '\t' << s.top();
            s.pop();
        }
        cout << '\n';
}
int main()
{
        stack <int> s;
        s.push(10);
        s.push(30);
        s.push(20);
        s.push(5);
        s.push(1);
        cout << "The stack is : ";
        showstack(s);
        cout << "\ns.size() : " << s.size();
        cout << "\ns.top() : " << s.size();
        cout << "\ns.pop() : ";
        s.pop();
        showstack(s);
        return 0;
}
```

OUTPUT :



Problem :-Reverse a Queue

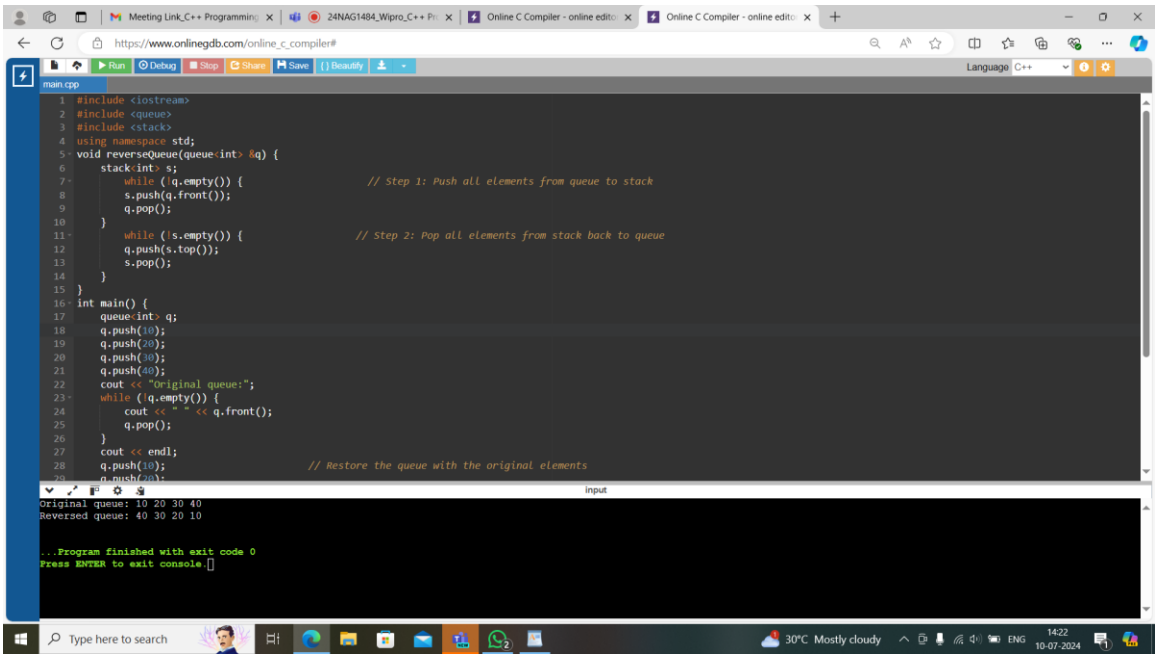
Description:

Implement a function to reverse the elements of a queue using a stack.

```
#include <iostream>

#include <queue>

#include <stack>

using namespace std;

void reverseQueue(queue<int> &q) {

    stack<int> s;

        while (!q.empty()) {                    // Step 1: Push all elements from queue to stack

        s.push(q.front());

        q.pop();

    }

        while (!s.empty()) {                    // Step 2: Pop all elements from stack back to queue

        q.push(s.top());
```

```
            s.pop();
        }
}
int main() {
        queue<int> q;
        q.push(10);
        q.push(20);
        q.push(30);
        q.push(40);
        cout << "Original queue:";
        while (!q.empty()) {
            cout << " " << q.front();
            q.pop();
        }
        cout << endl;
        q.push(10);                                    // Restore the queue with the original elements
        q.push(20);
        q.push(30);
        q.push(40);
        reverseQueue(q);                          // Reverse the queue
        cout << "Reversed queue:";
        while (!q.empty()) {
            cout << " " << q.front();
            q.pop();
        }
        cout << endl;
        return 0;
}
```

OUTPUT :-



Maximum Element in Stack

Description:

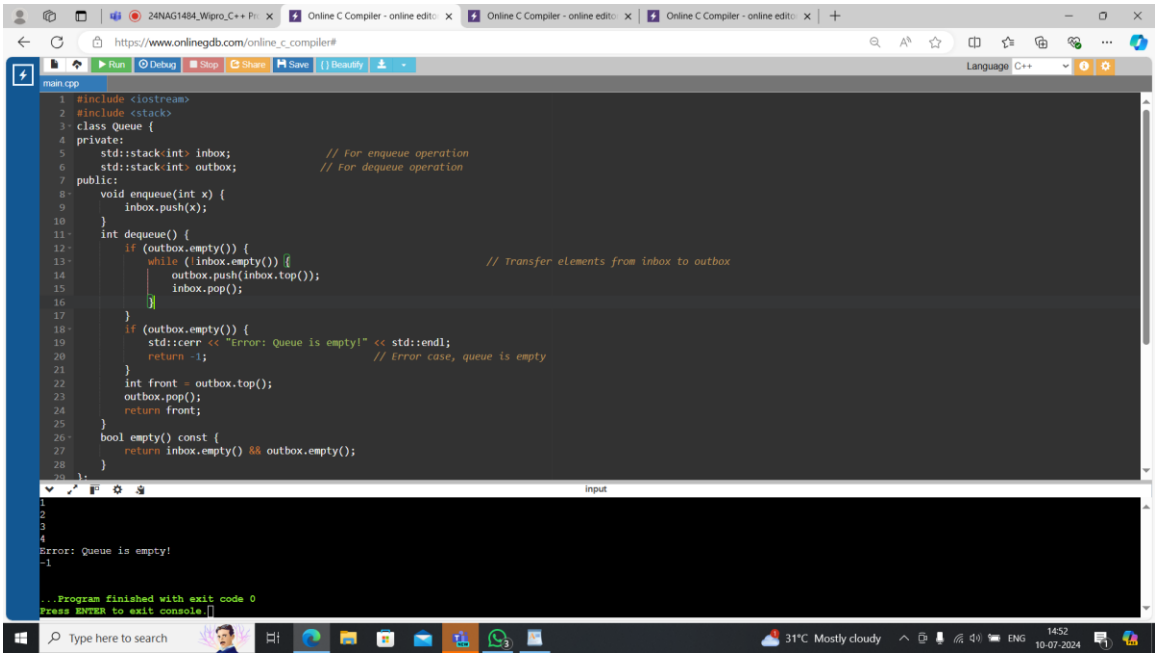
Design a stack that supports push, pop, and retrieving the maximum element in constant time.

```
#include <iostream>
#include <stack>
class Queue {
private:
    std::stack<int> inbox;                  // For enqueue operation
    std::stack<int> outbox;                 // For dequeue operation
public:
    void enqueue(int x) {
        inbox.push(x);
    }
    int dequeue() {
        if (outbox.empty()) {
```

```
            while (!inbox.empty()) {                                    // Transfer elements
from inbox to outbox
                outbox.push(inbox.top());
                inbox.pop();
            }
        }
        if (outbox.empty()) {
            std::cerr << "Error: Queue is empty!" << std::endl;
            return -1;                                  // Error case, queue is empty
        }
        int front = outbox.top();
        outbox.pop();
        return front;
    }
    bool empty() const {
        return inbox.empty() && outbox.empty();
    }
};
int main() {
    Queue q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);
    std::cout << q.dequeue() << std::endl;    // Output: 1
    std::cout << q.dequeue() << std::endl;    // Output: 2
    q.enqueue(4);
    std::cout << q.dequeue() << std::endl;    // Output: 3
    std::cout << q.dequeue() << std::endl;    // Output: 4
```

```
    std::cout << q.dequeue() << std::endl;    // Output: Error message, queue is empty
    return 0;
}
```

OUTPUT :-



Circular Queue Implementation

Description:

Implement a circular queue using an array. The queue should support enqueue, dequeue, and front operations.

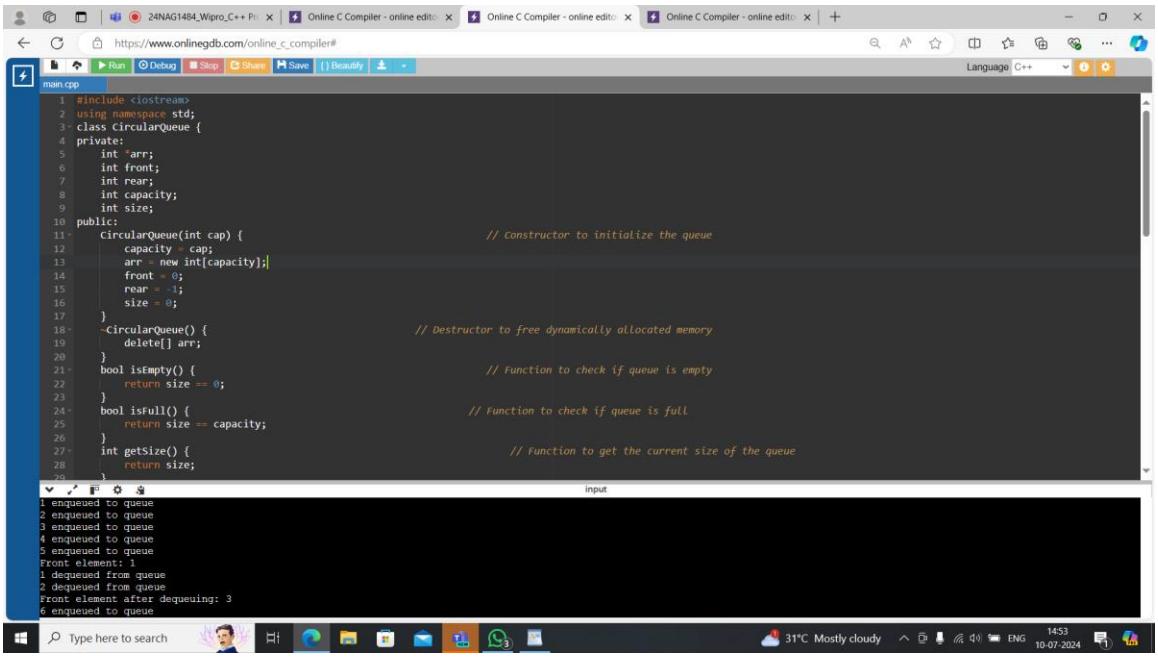
```
#include <iostream>
using namespace std;
class CircularQueue {
private:
    int *arr;
    int front;
    int rear;
    int capacity;
```

```
    int size;
public:
    CircularQueue(int cap) {                                    // Constructor to initialize the queue
        capacity = cap;
        arr = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }
    ~CircularQueue() {                                          // Destructor to free dynamically allocated memory
        delete[] arr;
    }
    bool isEmpty() {                                            // Function to check if queue is empty
        return size == 0;
    }
    bool isFull() {                                             // Function to check if queue is full
        return size == capacity;
    }
    int getSize() {                                             // Function to get the current size of the queue
        return size;
    }
    void enqueue(int value) {                                   // Function to enqueue an element to the queue
        if (isFull()) {
            cout << "Queue Overflow\n";
```

```
            return;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        size++;
        cout << value << " enqueued to queue\n";
    }
    void dequeue() {                                                                    //
Function to dequeue an element from the queue
        if (isEmpty()) {
            cout << "Queue Underflow\n";
            return;
        }

        int removedValue = arr[front];
        front = (front + 1) % capacity;
        size--;
        cout << removedValue << " dequeued from queue\n";
    }
    int frontElement() {                                                        // Function to
get the front element of the queue
        if (isEmpty()) {
            cout << "Queue is empty\n";
            return -1;
        }

        return arr[front];
    }
```

```
};
int main() {
    CircularQueue queue(5);
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
    queue.enqueue(4);
    queue.enqueue(5);
    cout << "Front element: " << queue.frontElement() << endl;
    queue.dequeue();
    queue.dequeue();
    cout << "Front element after dequeuing: " << queue.frontElement() << endl;
    queue.enqueue(6);
    queue.enqueue(7);
    queue.dequeue();
    queue.dequeue();
    queue.dequeue();
    queue.dequeue();
    queue.dequeue();
    return 0;
}
```

OUTPUT :-



Sort a Stack

Description:

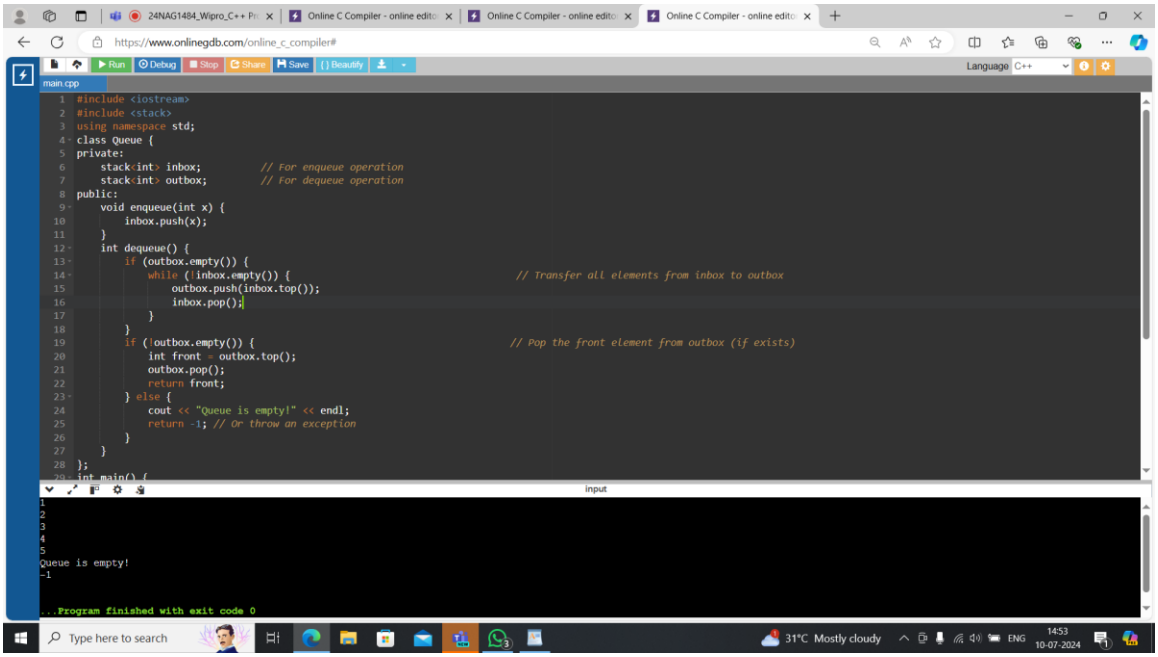
Write a function to sort a stack such that the smallest items are on the top.

```
#include <iostream>
#include <stack>
using namespace std;
class Queue {
private:
    stack<int> inbox;          // For enqueue operation
    stack<int> outbox;         // For dequeue operation
public:
    void enqueue(int x) {
        inbox.push(x);
    }
    int dequeue() {
        if (outbox.empty()) {
```

```
            while (!inbox.empty()) {                                    // Transfer all elements from inbox to outbox
                outbox.push(inbox.top());
                inbox.pop();
            }
        }
        if (!outbox.empty()) {                                          // Pop the front element from outbox (if exists)
            int front = outbox.top();
            outbox.pop();
            return front;
        } else {
            cout << "Queue is empty!" << endl;
            return -1; // Or throw an exception
        }
    }
};
int main() {
    Queue q;
    q.enqueue(1);                       // Enqueue some elements
    q.enqueue(2);
    q.enqueue(3);
    cout << q.dequeue() << endl;                                        // Dequeue elements
    cout << q.dequeue() << endl;
    q.enqueue(4);                                                       // Enqueue more elements
    q.enqueue(5);
    cout << q.dequeue() << endl; // Output: 3                           // Dequeue remaining elements
    cout << q.dequeue() << endl; // Output: 4
```

```
    cout << q.dequeue() << endl; // Output: 5
    cout << q.dequeue() << endl; // Output: Queue is empty!
    return 0;
}
```

OUTPUT :-



STD::LIST :-

```
#include <iostream>
#include <list>
int main() {
    std::list<int> myList;                                  // Create a list
    myList.push_back(10);                                   // Insert elements at the end
    myList.push_back(20);
    myList.push_back(30);

    myList.push_front(5);                                   // Insert elements at the front
    myList.push_front(1);
```

```
std::cout << "List after push_back and push_front: ";                    // Display elements
for (int val : myList) {
        std::cout << val << " ";
}
std::cout << std::endl;
auto it = myList.begin();                                                 // Insert element at a specific position
std::advance(it, 2);
myList.insert(it, 15);
std::cout << "List after insert: ";
for (int val : myList) {
        std::cout << val << " ";
}
std::cout << std::endl;
it = myList.begin();                                                      // Erase element at a specific position
std::advance(it, 3);
myList.erase(it);
std::cout << "List after erase: ";
for (int val : myList) {
        std::cout << val << " ";
}
std::cout << std::endl;
myList.remove(10);                                                        // Remove elements by value
std::cout << "List after remove: ";
for (int val : myList) {
        std::cout << val << " ";
}
std::cout << std::endl;
myList.remove_if([](int n) { return n < 10; });                           // Remove elements based on a
```

```
condition
    std::cout << "List after remove_if: ";
    for (int val : myList) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    myList.sort();                                          // Sorting the list
    std::cout << "List after sort: ";
    for (int val : myList) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    myList.reverse();                                       // Reversing the list
    std::cout << "List after reverse: ";
    for (int val : myList) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    std::list<int> otherList = {40, 50, 60};                // Merging two lists
    myList.merge(otherList);
    std::cout << "List after merge: ";
    for (int val : myList) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    myList.clear();                                         // Clearing the list
    std::cout << "List after clear: ";
    for (int val : myList) {
```

```
        std::cout << val << " ";
    }
    std::cout << std::endl;
    if (myList.empty()) {                                   // Checking if the list is empty
        std::cout << "List is empty." << std::endl;
    }
    myList.push_back(100);                                  // Adding elements again
    myList.push_back(200);
    std::cout << "Front element: " << myList.front() << std::endl;          // Accessing front and back elements
    std::cout << "Back element: " << myList.back() << std::endl;
    return 0;
}
```

OUTPUT :-