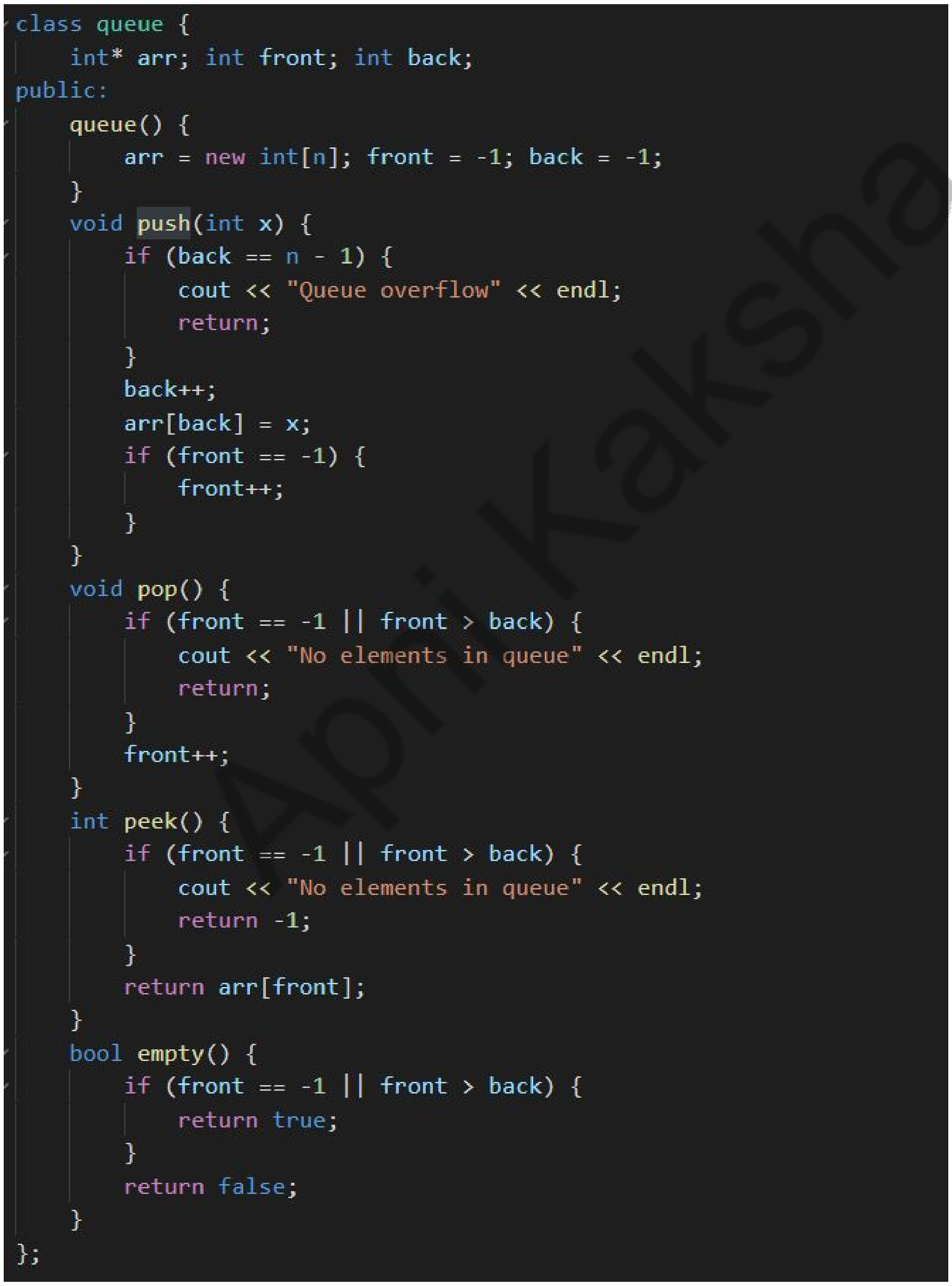
Queues

Queue is a linear data structure that stores the data in First In First Out (FIFO) fashion.

Queue using Arrays:

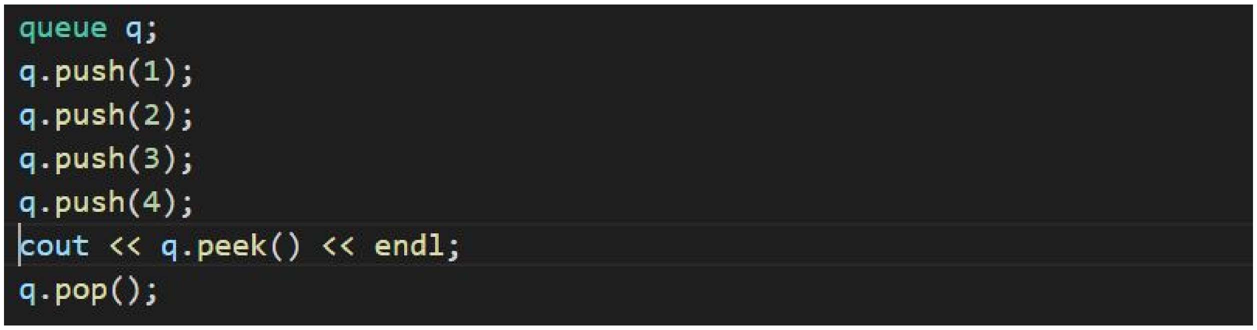
Idea: Make an array. Add from the end of the array. Delete from the end.

Implementation:



```
class queue {
    int* arr; int front; int back;
public:
    queue() {
        arr = new int[n]; front = -1; back = -1;
    }
    void push(int x) {
        if (back == n - 1) {
            cout << "Queue overflow" << endl;
            return;
        }
        back++;
        arr[back] = x;
        if (front == -1) {
            front++;
        }
    }
    void pop() {
        if (front == -1 || front > back) {
            cout << "No elements in queue" << endl;
            return;
        }
        front++;
    }
    int peek() {
        if (front == -1 || front > back) {
            cout << "No elements in queue" << endl;
            return -1;
        }
        return arr[front];
    }
    bool empty() {
        if (front == -1 || front > back) {
            return true;
        }
        return false;
    }
};
```

Usage:

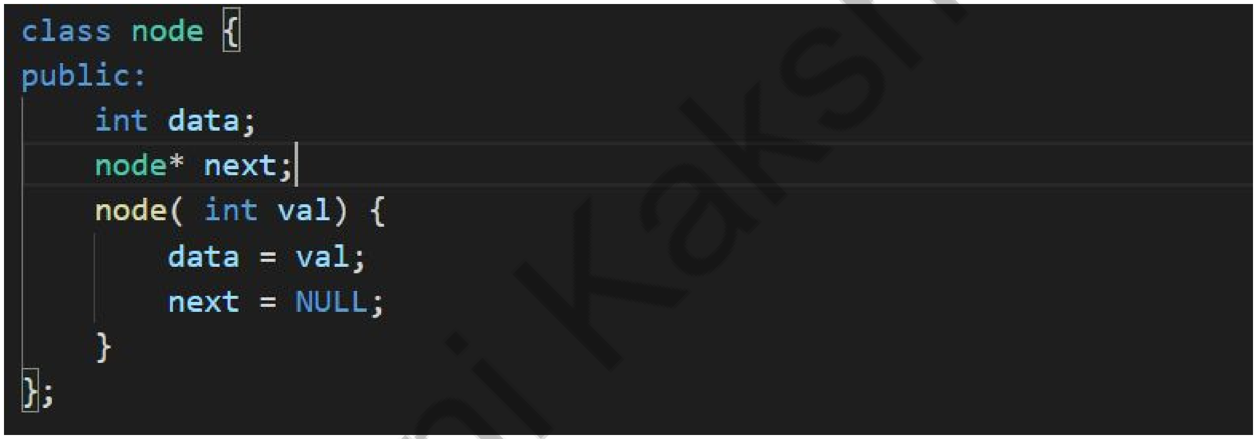


```
queue q;
q.push(1);
q.push(2);
q.push(3);
q.push(4);
cout << q.peek() << endl;
q.pop();
```

Note: But it is restricted with the size of the array.

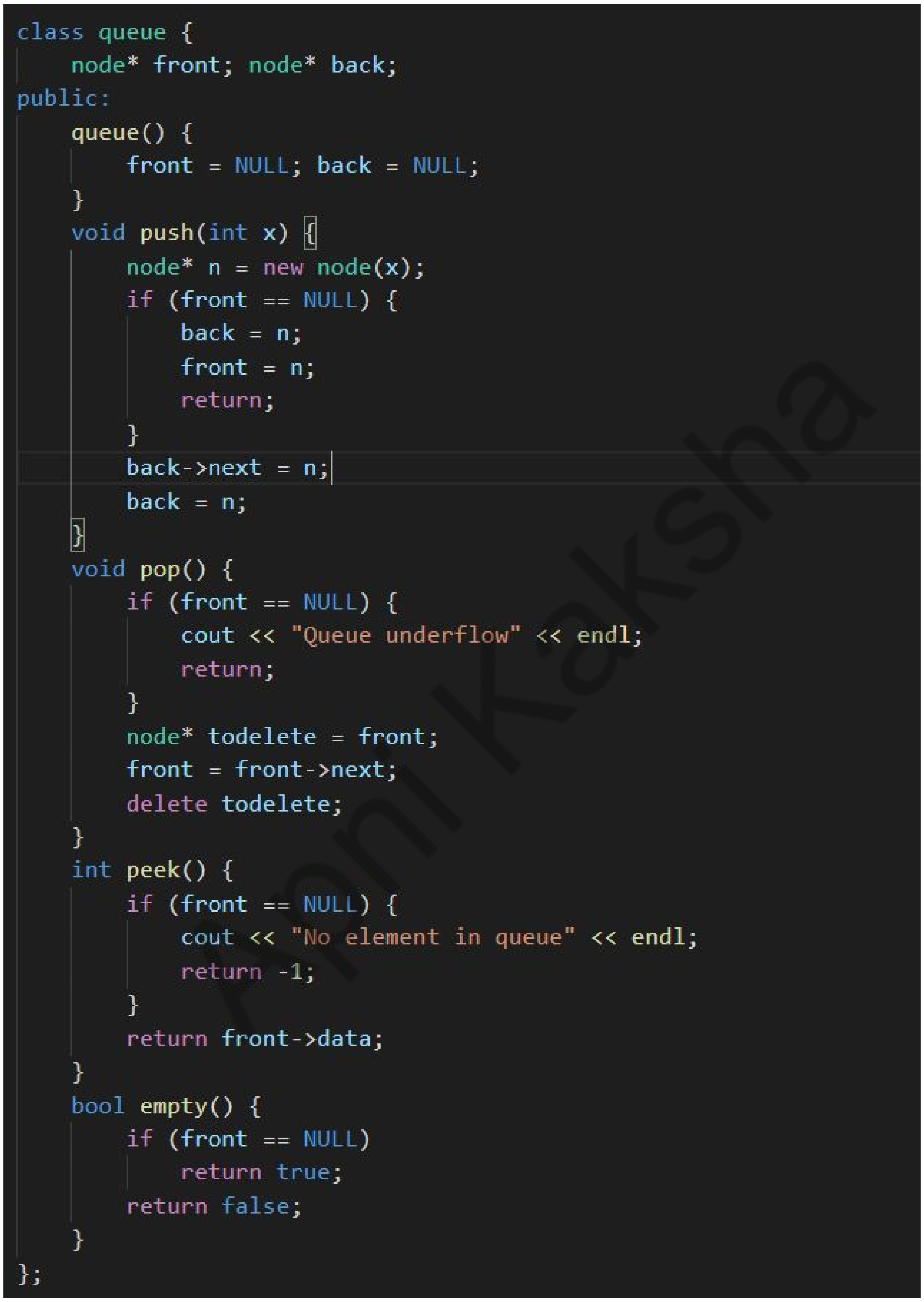
Queue using LinkedList

First, we need to define the basic entity of our queue node.



```
class node {
public:
    int data;
    node* next;
    node( int val) {
        data = val;
        next = NULL;
    }
};
```

Main Class Definition:



```
class queue {
    node* front; node* back;
public:
    queue() {
        front = NULL; back = NULL;
    }
    void push(int x) {
        node* n = new node(x);
        if (front == NULL) {
            back = n;
            front = n;
            return;
        }
        back->next = n;
        back = n;
    }
    void pop() {
        if (front == NULL) {
            cout << "Queue underflow" << endl;
            return;
        }
        node* todelete = front;
        front = front->next;
        delete todelete;
    }
    int peek() {
        if (front == NULL) {
            cout << "No element in queue" << endl;
            return -1;
        }
        return front->data;
    }
    bool empty() {
        if (front == NULL)
            return true;
        return false;
    }
};
```

Stack using queues

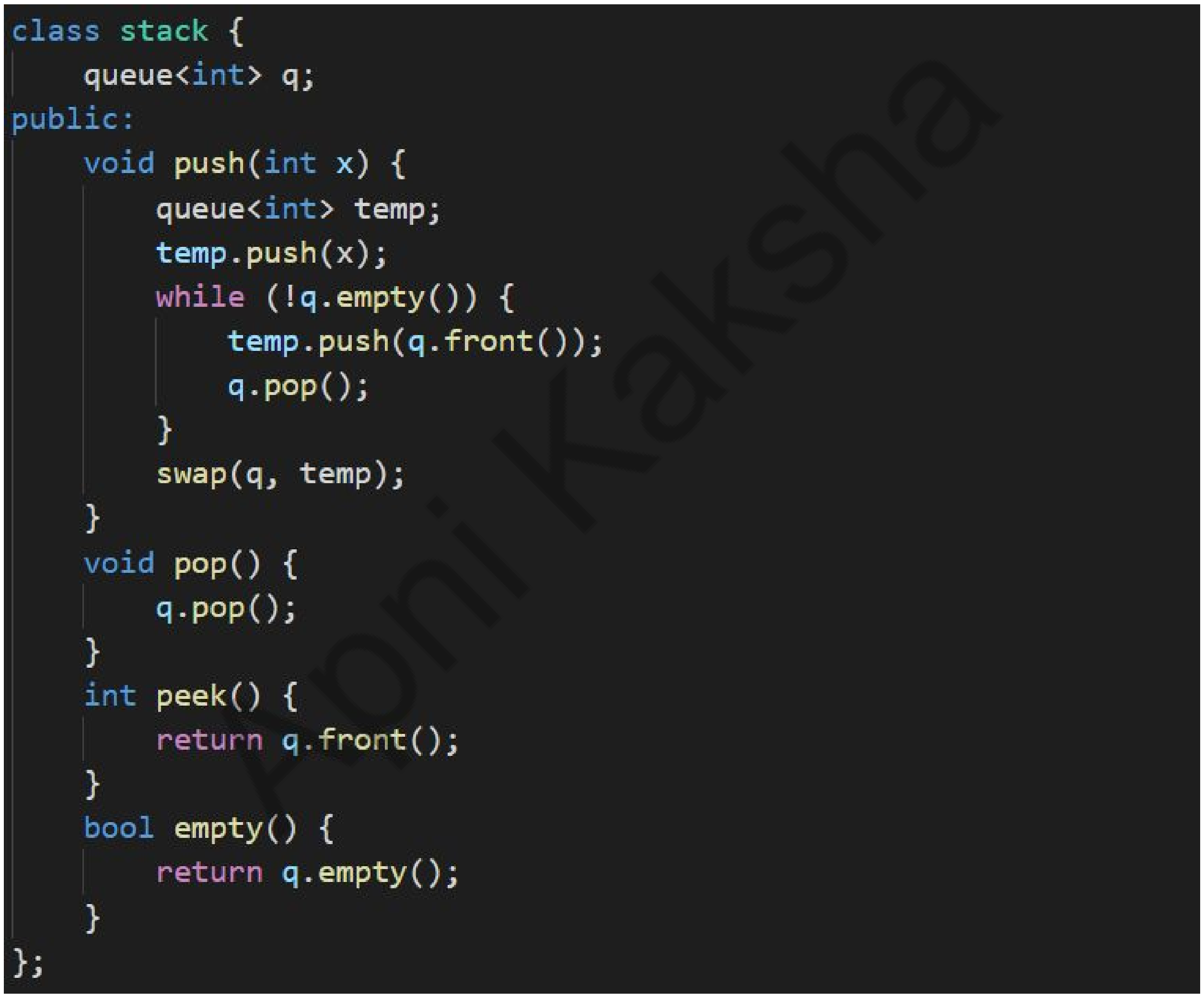
Idea:

There are two types:

1. Push costly
2. Pop costly

The below code is Push costly.

For each push, we use a temporary queue, enqueue x first, and then enqueue the elements from the other queue.



```
class stack {
    queue<int> q;
public:
    void push(int x) {
        queue<int> temp;
        temp.push(x);
        while (!q.empty()) {
            temp.push(q.front());
            q.pop();
        }
        swap(q, temp);
    }
    void pop() {
        q.pop();
    }
    int peek() {
        return q.front();
    }
    bool empty() {
        return q.empty();
    }
};
```

Queue using stacks

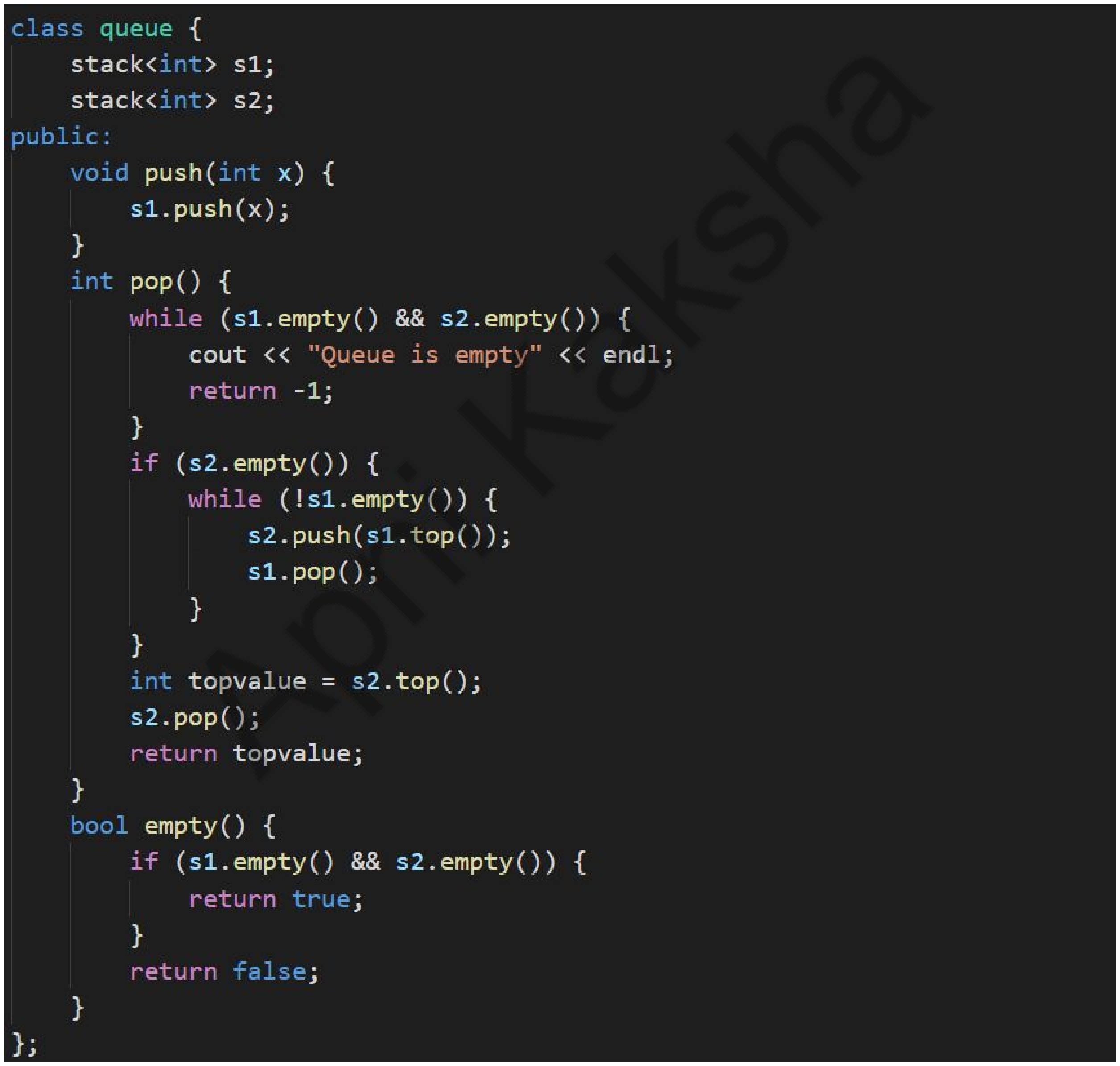
Idea:

There are two types:

1. Enqueue costly
2. Dequeue costly

The below code is Dequeue/ pop costly.

Idea: we keep 2 stacks, we enqueue in 1 stack. For dequeue, we do shift elements into another stack.



```
class queue {
    stack<int> s1;
    stack<int> s2;
public:
    void push(int x) {
        s1.push(x);
    }
    int pop() {
        while (s1.empty() && s2.empty()) {
            cout << "Queue is empty" << endl;
            return -1;
        }
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        int topvalue = s2.top();
        s2.pop();
        return topvalue;
    }
    bool empty() {
        if (s1.empty() && s2.empty()) {
            return true;
        }
        return false;
    }
};
```

Online Judges for the problems:

1. Stacks using queue.
2. Queue using stack.