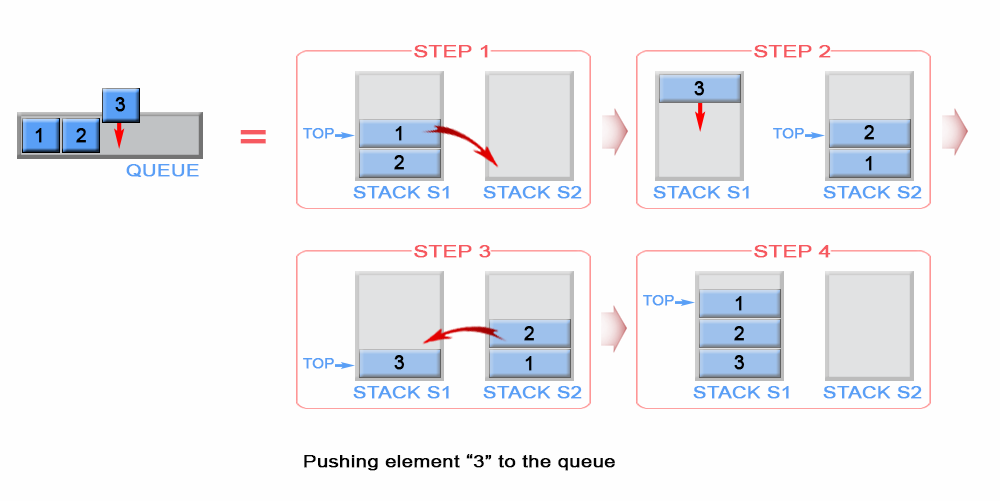
Queue is **FIFO** (first in - first out) data structure, in which the elements are inserted from one side - rear and removed from the other - front. The most intuitive way to implement it is with linked lists, but this article will introduce another approach using stacks. Stack is **LIFO** (last in - first out) data structure, in which elements are added and removed from the same end, called top. To satisfy **FIFO** property of a queue we need to keep two stacks. They serve to reverse arrival order of the elements and one of them store the queue elements in their final order.

Approach #1 (Two Stacks) Push - O(n)*O*(*n*) per operation, Pop - O(1)*O*(1) per operation.

**Algorithm**

**Push**

A queue is FIFO (first-in-first-out) but a stack is LIFO (last-in-first-out). This means the newest element must be pushed to the bottom of the stack. To do so we first transfer all s1 elements to auxiliary stack s2. Then the newly arrived element is pushed on top of s2 and all its elements are popped and pushed to s1.



*Figure 1. Push an element in queue*

**Java**

private int front;

public void push(int x) {

if (s1.empty())

front = x;

while (!s1.isEmpty())

s2.push(s1.pop());

s2.push(x);

while (!s2.isEmpty())

s1.push(s2.pop());

}

**Complexity Analysis**

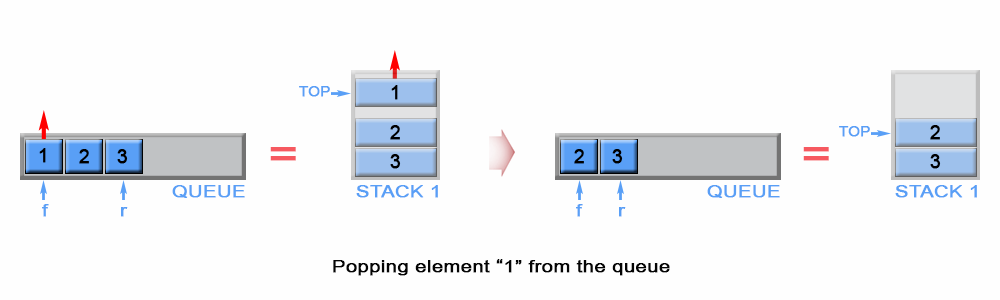
* Time complexity : O(n)*O*(*n*).

Each element, with the exception of the newly arrived, is pushed and popped twice. The last inserted element is popped and pushed once. Therefore this gives 4 n + 24*n*+2 operations where n*n* is the queue size. The push and pop operations have O(1)*O*(1) time complexity.

* Space complexity : O(n)*O*(*n*). We need additional memory to store the queue elements

**Pop**

The algorithm pops an element from the stack s1, because s1 stores always on its top the first inserted element in the queue. The front element of the queue is kept as front.



*Figure 2. Pop an element from queue*

**Java**

// Removes the element from the front of queue.

public void pop() {

s1.pop();

if (!s1.empty())

front = s1.peek();

}

**Complexity Analysis**

* Time complexity : O(1)*O*(1).
* Space complexity : O(1)*O*(1).

**Empty**

Stack s1 contains all stack elements, so the algorithm checks s1 size to return if the queue is empty.

// Return whether the queue is empty.

public boolean empty() {

return s1.isEmpty();

}

Time complexity : O(1)*O*(1).

Space complexity : O(1)*O*(1).

**Peek**

The front element is kept in constant memory and is modified when we push or pop an element.

// Get the front element.

public int peek() {

return front;

}

Time complexity : O(1)*O*(1). The front element has been calculated in advance and only returned in peek operation.

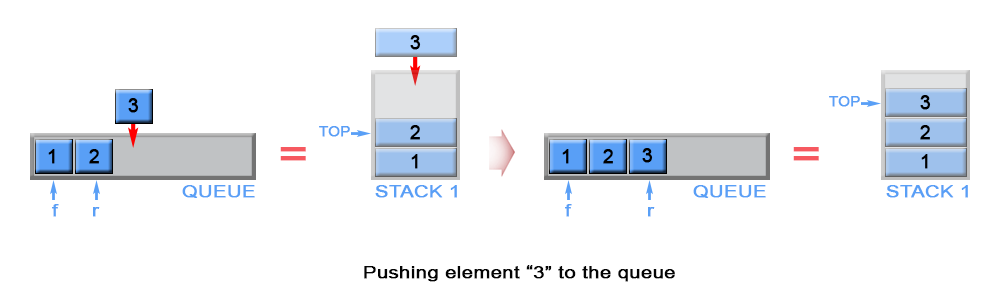
Space complexity : O(1)*O*(1).

Approach #2 (Two Stacks) Push - O(1)*O*(1) per operation, Pop - Amortized O(1)*O*(1) per operation.

**Algorithm**

**Push**

The newly arrived element is always added on top of stack s1 and the first element is kept as front queue element



*Figure 3. Push an element in queue*

**Java**

private Stack<Integer> s1 = new Stack<>();

private Stack<Integer> s2 = new Stack<>();

// Push element x to the back of queue.

public void push(int x) {

if (s1.empty())

front = x;

s1.push(x);

}

**Complexity Analysis**

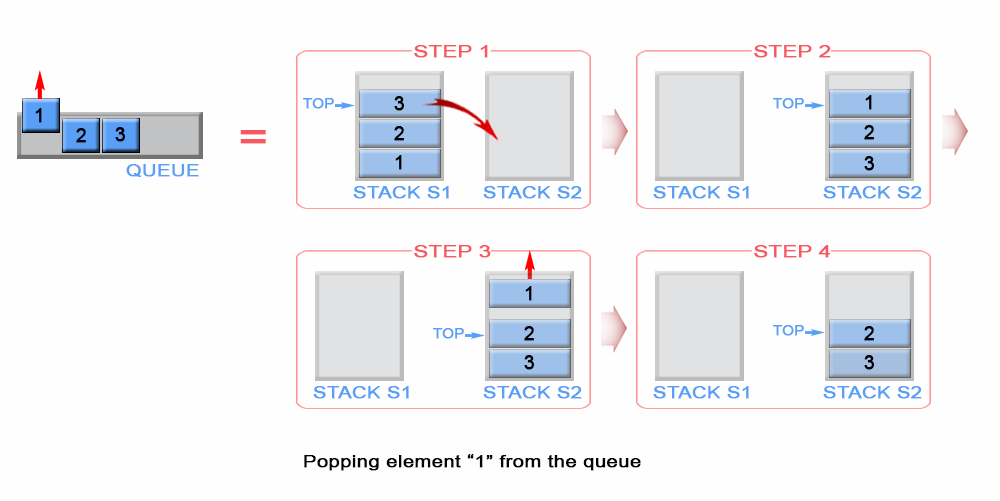
* Time complexity : O(1)*O*(1).

Аppending an element to a stack is an O(1) operation.

* Space complexity : O(n)*O*(*n*). We need additional memory to store the queue elements

**Pop**

We have to remove element in front of the queue. This is the first inserted element in the stack s1 and it is positioned at the bottom of the stack because of stack's LIFO (last in - first out) policy. To remove the bottom element from s1, we have to pop all elements from s1 and to push them on to an additional stack s2, which helps us to store the elements of s1 in reversed order. This way the bottom element of s1 will be positioned on top of s2 and we can simply pop it from stack s2. Once s2 is empty, the algorithm transfer data from s1 to s2 again.



*Figure 4. Pop an element from stack*

**Java**

// Removes the element from in front of queue.

public void pop() {

if (s2.isEmpty()) {

while (!s1.isEmpty())

s2.push(s1.pop());

}

s2.pop();

}

**Complexity Analysis**

* Time complexity: Amortized O(1)*O*(1), Worst-case O(n)*O*(*n*).

In the worst case scenario when stack s2 is empty, the algorithm pops n*n* elements from stack s1 and pushes n*n* elements to s2, where n*n* is the queue size. This gives 2n2*n* operations, which is O(n)*O*(*n*). But when stack s2 is not empty the algorithm has O(1)*O*(1) time complexity. So what does it mean by Amortized O(1)*O*(1)? Please see the next section on Amortized Analysis for more information.

* Space complexity : O(1)*O*(1).

**Amortized Analysis**

Amortized analysis gives the average performance (over time) of each operation in the worst case. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus amortizing its cost.

Consider this example where we start with an empty queue with the following sequence of operations applied:

push\_1, push\_2, \ldots, push\_n, pop\_1,pop\_2 \ldots, pop\_n*push*1​,*push*2​,…,*pushn*​,*pop*1​,*pop*2​…,*popn*​

The worst case time complexity of a single pop operation is O(n)*O*(*n*). Since we have n*n* pop operations, using the worst-case per operation analysis gives us a total of O(n^2)*O*(*n*2) time.

However, in a sequence of operations the worst case does not occur often in each operation - some operations may be cheap, some may be expensive. Therefore, a traditional worst-case per operation analysis can give overly pessimistic bound. For example, in a dynamic array only some inserts take a linear time, though others - a constant time.

In the example above, the number of times pop operation can be called is limited by the number of push operations before it. Although a single pop operation could be expensive, it is expensive only once per n times (queue size), when s2 is empty and there is a need for data transfer between s1 and s2. Hence the total time complexity of the sequence is : n (for push operations) + 2\*n (for first pop operation) + n - 1 ( for pop operations) which is O(2\*n)*O*(2∗*n*).This gives O(2n/2n)*O*(2*n*/2*n*) = O(1)*O*(1) average time per operation.

**Empty**

Both stacks s1 and s2 contain all stack elements, so the algorithm checks s1 and s2 size to return if the queue is empty.

// Return whether the queue is empty.

public boolean empty() {

return s1.isEmpty() && s2.isEmpty();

}

Time complexity : O(1)*O*(1).

Space complexity : O(1)*O*(1).

**Peek**

The front element is kept in constant memory and is modified when we push an element. When s2 is not empty, front element is positioned on the top of s2

// Get the front element.

public int peek() {

if (!s2.isEmpty()) {

return s2.peek();

}

return front;

}

Time complexity : O(1)*O*(1).

The front element was either previously calculated or returned as a top element of stack s2. Therefore complexity is O(1)*O*(1)

Space complexity : O(1)*O*(1).