

1. Implement Queue using Stack

232. Implement Queue using Stacks Solved

Easy Topics Companies

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.

Notes:

- You must use **only** standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

Input
["MyQueue", "push", "push", "peek", "pop", "empty"]

Output
[null, null, null, 1, null, true]

```
1 class MyQueue {
2 private:
3     stack<int> input;
4     stack<int> output;
5
6 public:
7     MyQueue() {}
8
9     void push(int x) {
10         input.push(x);
11     }
12
13     int pop() {
14         peek();
15         int val = output.top();
16         output.pop();
17         return val;
18     }
19 }
```

Accepted Runtime: 0 ms

Case 1

Input
["MyQueue", "push", "push", "peek", "pop", "empty"]

3. Implement Min Stack using Two Stacks

155. Min Stack Solved

Medium Topics Companies Hint

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

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with $O(1)$ time complexity for each function.

Example 1:

Input
["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]
[[], [-2], [0], [-3], [], [], [], []]

Output
[null, null, null, null, -3, null, 0, -2]

```
1 class MinStack {
2 private:
3     vector<vector<int>>> st;
4
5 public:
6     MinStack() {}
7
8
9     void push(int val) {
10         int min_val = getMin();
11         if (st.empty() || min_val > val) {
12             min_val = val;
13         }
14         st.push_back({val, min_val});
15     }
16
17     void pop() {
18         st.pop_back();
19     }
20 }
```

Accepted Runtime: 0 ms

Case 1

Input
["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]

6. Implement BST (Inorder Traversal) using Stack (Iterative DFS)

94. Binary Tree Inorder Traversal

Given the `root` of a binary tree, return the *inorder traversal* of its nodes' values.

Example 1:

Input: `root = [1,null,2,3]`

Output: `[1,3,2]`

Explanation:

```
graph TD
    1((1)) --- 3((3))
    1 --- 2((2))
    2 --- 3
```

Code:

```
C++  
class Solution {  
public:  
    vector<int> inorderTraversal(TreeNode* root) {  
        vector<int> res;  
        inorder(root, res);  
        return res;  
    }  
private:  
    void inorder(TreeNode* node, vector<int>& res) {  
        if (!node) return;  
        inorder(node->left, res);  
        res.push_back(node->val);  
        inorder(node->right, res);  
    }  
};
```

Testcase: Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4

Input: `root = [1,null,2,3]`

8. Implement Stack using Queue

225. Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element `x` to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a queue, which means that only `push` to back, `peek/pop` from front, `size` and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

Input: `["MyStack", "push", "push", "top", "pop", "empty"]`

Code:

```
C++  
int top() {  
    while (q1.size() > 1) {  
        q2.push(q1.front());  
        q1.pop();  
    }  
    int topVal = q1.front();  
    q2.push(q1.front());  
    q1.pop();  
    std::swap(q1, q2);  
    return topVal;  
}  
bool empty() {  
    return q1.empty();  
};
```

Testcase: Accepted Runtime: 0 ms

Case 1

Input: `["MyStack", "push", "push", "top", "pop", "empty"]`

10. Implement Circular Queue using Queue

622. Design Circular Queue

Medium Topics Companies

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the `MyCircularQueue` class:

- `MyCircularQueue(k)` Initializes the object with the size of the queue to be `k`.
- `int Front()` Gets the front item from the queue. If the queue is empty, return `-1`.
- `int Rear()` Gets the last item from the queue. If the queue is empty, return `-1`.
- `boolean enqueue(int value)` Inserts an element into the circular queue. Return `true` if the operation is successful.
- `boolean dequeue()` Deletes an element from the circular queue. Return `true` if the operation is successful.
- `boolean isEmpty()` Checks whether the circular queue is empty or not.
- `boolean isFull()` Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

```
58 // Get element at the rear
59 int Rear() {
60     if (rear == -1) {
61         return -1; // If queue is empty
62     }
63     return queue[rear];
64 }
65
66 // Check if queue is empty
67 bool isEmpty() {
68     return front == -1;
69 }
70
71 // Check if queue is full
72 bool isFull() {
73     return (rear + 1) % n == front;
74 }
75 }
```

Accepted Runtime: 0 ms

Case 1

Input

["MyCircularQueue","enqueue","enqueue","enqueue","enqueue","Rear","isFull","dequeue","enqueue","Rear"]

11. Implement BST Level Order Traversal using Queue (BFS)

102. Binary Tree Level Order Traversal

Medium Topics Companies Hint

Given the `root` of a binary tree, return the *level order traversal* of its nodes' values. (i.e., from left to right, level by level).

Example 1:

```
graph TD
    3((3)) --> 9((9))
    3 --> 20((20))
    9 --> 15((15))
    20 --> 7((7))
```

Input: `root = [3,9,20,null,null,15,7]`
Output: `[[3],[9,20],[15,7]]`

```
1 class Solution {
2 public:
3     vector<vector<int>> levelOrder(TreeNode* root) {
4         vector<vector<int>> ans;
5         if (!root) return ans;
6
7         queue<TreeNode*> q;
8         q.push(root);
9
10        while (!q.empty()) {
11            int level_size = q.size();
12            vector<int> level;
13
14            for (int i = 0; i < level_size; ++i) {
15                TreeNode* node = q.front();
16                q.pop();
17                level.push_back(node->val);
18                ...
19            }
20        }
21    }
22 }
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

root = [3,9,20,null,null,15,7]

13. Implement Stack using an Array

The screenshot shows a web browser with multiple tabs open, including 'leetcode.com/problems/build-an-array-with-stack-operations/description/'. The main content area displays the problem description for '1441. Build an Array With Stack Operations'. The problem is categorized as 'Medium' and includes a 'Hint' section. The description states: 'You are given an integer array target and an integer n. You have an empty stack with the two following operations: • "Push": pushes an integer to the top of the stack. • "Pop": removes the integer on the top of the stack. You also have a stream of the integers in the range [1, n]. Use the two stack operations to make the numbers in the stack (from the bottom to the top) equal to target. You should follow the following rules: • If the stream of the integers is not empty, pick the next integer from the stream and push it to the top of the stack. • If the stack is not empty, pop the integer at the top of the stack. • If, at any moment, the elements in the stack (from the bottom to the top) are equal to target, do not read new integers from the stream and do not do more operations on the stack. Return the stack operations needed to build target following the mentioned rules. If there are multiple valid answers, return any of them.' Below the description, there is a 'Code' editor with a C++ solution. The code is as follows:

```
6
7
8     for (int i = 0; i < target.size(); i++) {
9         while (current < target[i]) {
10             // While the current number is less than the target number,
11             // push the current number and generate the "Push" operation.
12             result.push_back("Push");
13             result.push_back("Pop"); // After pushing, immediately pop.
14             current++;
15         }
16         // The current number matches the target number, so push it.
17         result.push_back("Push");
18         current++; // Move to the next number to be pushed.
19     }
20     return result;
21 }
22
23
```

 The code is saved and the test result is 'Accepted' with a runtime of 0 ms. The input is 'target = [1, 3]'.

1441. Build an Array With Stack Operations

Medium

You are given an integer array `target` and an integer `n`.

You have an empty stack with the two following operations:

- "Push": pushes an integer to the top of the stack.
- "Pop": removes the integer on the top of the stack.

You also have a stream of the integers in the range `[1, n]`.

Use the two stack operations to make the numbers in the stack (from the bottom to the top) equal to `target`. You should follow the following rules:

- If the stream of the integers is not empty, pick the next integer from the stream and push it to the top of the stack.
- If the stack is not empty, pop the integer at the top of the stack.
- If, at any moment, the elements in the stack (from the bottom to the top) are equal to `target`, do not read new integers from the stream and do not do more operations on the stack.

Return the stack operations needed to build `target` following the mentioned rules. If there are multiple valid answers, return **any of them**.

Example 1:

Input: `target = [1, 3]`

Output: `["Push", "Push", "Pop", "Push"]`

Explanation: Initially the stack is empty. The first stream of numbers is 1. A single push operation to the stack of 1 is enough to make `target` = [1], so we return `["Push"]`. The second stream of numbers is 3. A single push operation to the stack of 3 is enough to make `target` = [1, 3], so we return `["Push", "Push"]`. However, the second push operation can be skipped, because the stack already contains 1, 3. We can pop the 3, and then push 3, to make `target` = [1, 3]. So we return `["Push", "Pop", "Push"]`.

Runtime: 0 ms

Accepted

Case 1 Case 2 Case 3

Input

target =

[1, 3]