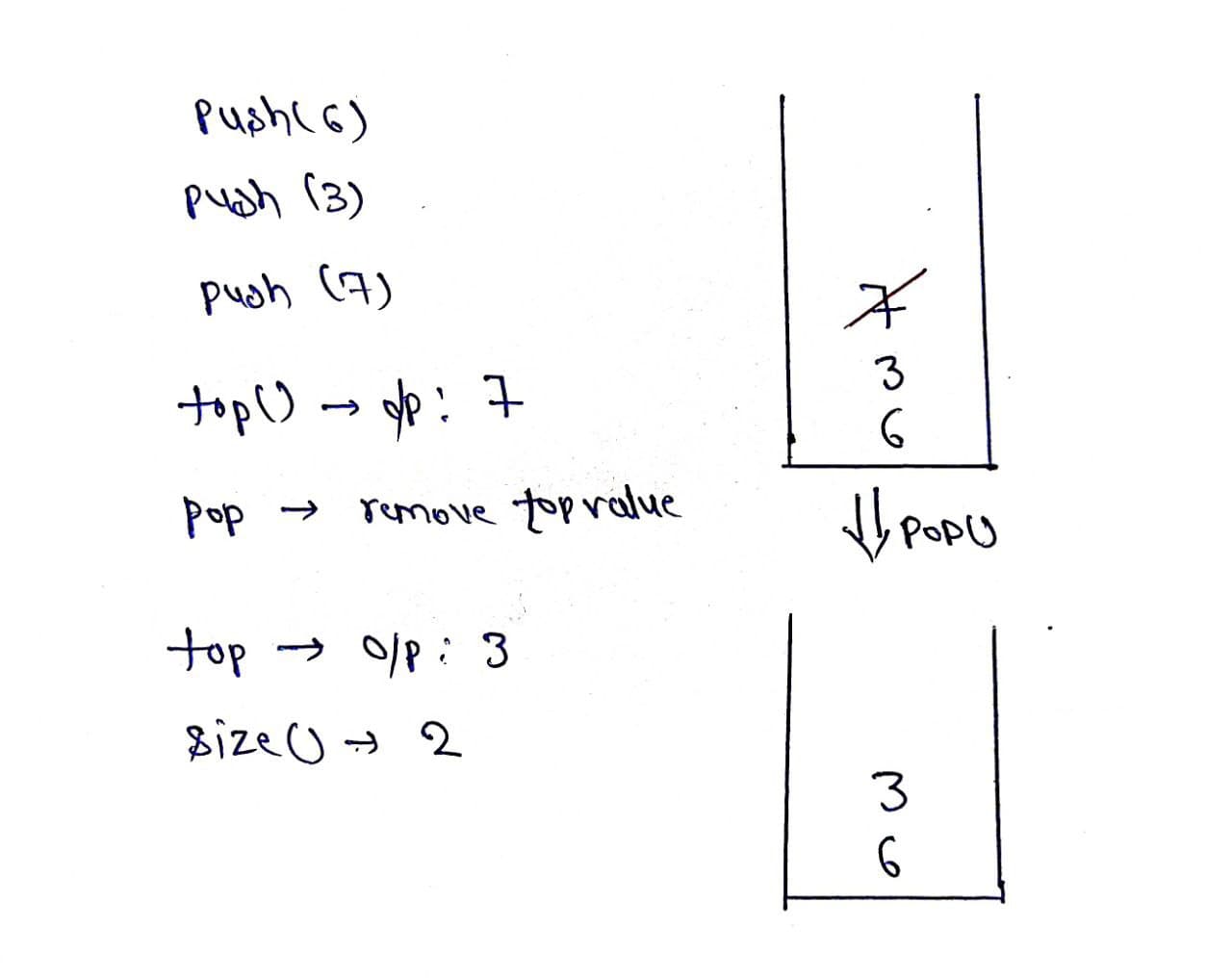
**Implement Stack using Array**

**Problem statement:** Implement a stack using an array.

**Note:** Stack is a data structure that follows the **Last In First Out (LIFO)** rule.

**Example:**



**Explanation**:

push(): Insert the element in the stack.

pop(): Remove and return the topmost element of the stack.

top(): Return the topmost element of the stack

size(): Return the number of remaining elements in the stack.

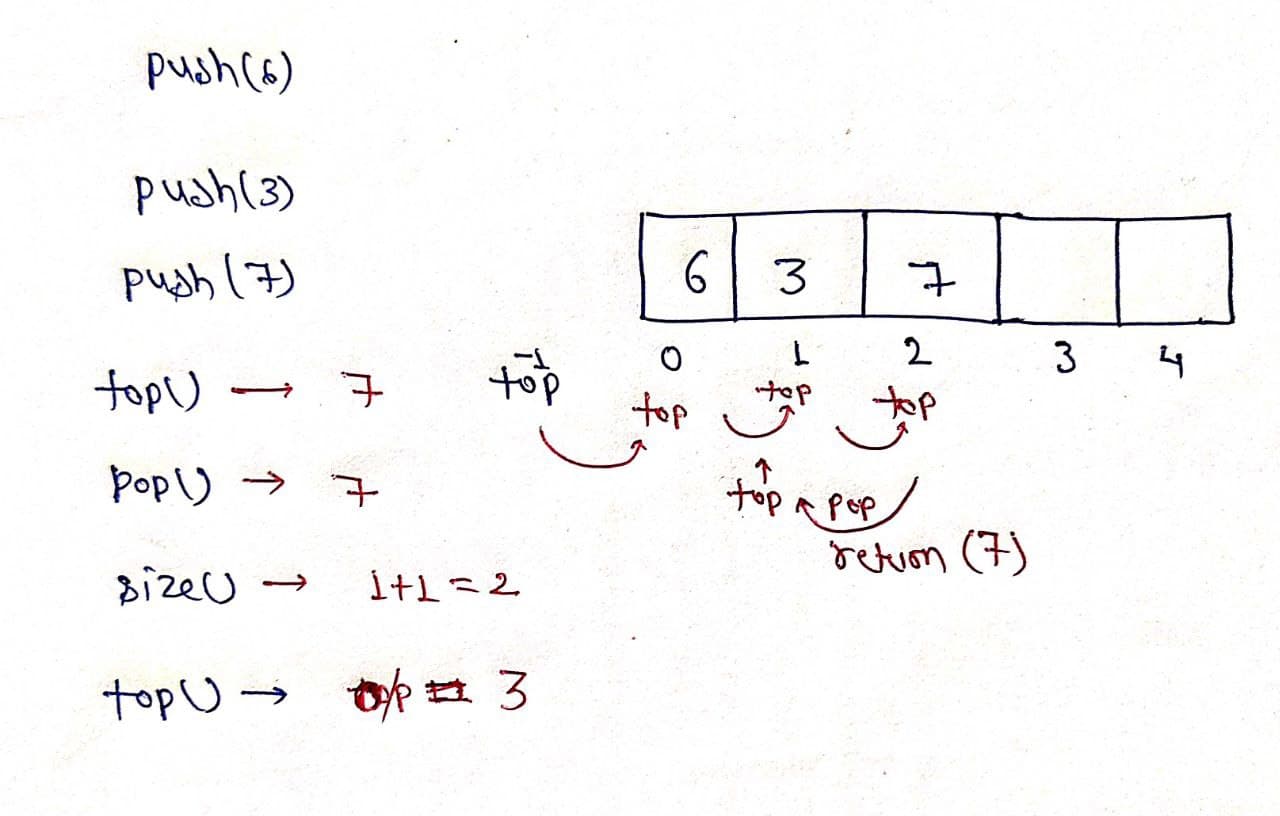
**Solution**

*Disclaimer*: *Don’t jump directly to the solution, try it out yourself first.*

**Intuition**: As we know stack works on the principle of last in first out, so we have to put elements in an array such that it keeps track of the most recently inserted element. Hence we can think of using a Top variable which will help in keeping track of recent elements inserted in the array.

**Approach**:

* Declare an array of particular size.
* Define a variable “Top” and initialize it as -1.
* push(x): insert element is the array by increasing top by one.
* pop(): check whether top is not equal to -1 if it is so, return top and decrease its value by one.
* size(): return top+1.



**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

public class tuf {

public static void main(String[] args) {

stack s = new stack();

s.push(6);

s.push(3);

s.push(7);

System.out.println("Top of the stack before deleting any element " + s.top());

System.out.println("Size of the stack before deleting any element " + s.size());

System.out.println("The element deleted is " + s.pop());

System.out.println("Size of the stack after deleting an element " + s.size());

System.out.println("Top of the stack after deleting an element " + s.top());

}

}

class stack {

int size = 10000;

int arr[] = new int[size];

int top = -1;

void push(int x) {

top++;

arr[top] = x;

}

int pop() {

int x = arr[top];

top--;

return x;

}

int top() {

return arr[top];

}

int size() {

return top + 1;

}

}

**Output:**

Top of the stack before deleting any element 7  
Size of the stack before deleting any element 3  
The element deleted is 7  
Size of the stack after deleting an element 2  
Top of the stack after deleting an element 3

**Time Complexity:** O(N)

**Space Complexity:** O(N)

# Implement Queue Using Array

**Problem Statement:** Implement Queue Data Structure using Array with all functions like pop, push, top, size, etc.

**Example:**

**Input**: push(4)

push(14)

push(24)

push(34)

top()

size()

pop()

**Output:**

The element pushed is 4

The element pushed is 14

The element pushed is 24

The element pushed is 34

The peek of the queue before deleting any element 4

The size of the queue before deletion 4

The first element to be deleted 4

The peek of the queue after deleting an element 14

The size of the queue after deleting an element 3

### ****Solution****

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Intuition**:

The intuition is to fill the array in a circular manner, (ie) after popping from the front, rather than moving all the elements towards the front. We can have 2 variables to keep track of the start and end indexes of the sequence. Mod addition is done to handle boundary conditions.

**Approach**:

The basic approach is to maintain two variables to point to the START and END of the filled elements in the array. START pointer is used to point to the starting index of the elements and the same case for the END pointer(ending index). Initially, both have value -1(indicating empty queue).

First, let’s see the implementation of the push function. Push basically inserts a new element at the end. So only the END variable is going to be incremented.

**Corner case 1:** What if we have no empty places in the array? So, first check that, if we don’t have we exit, in the other case we increment the START variable and put the new element.

**Corner case 2:** What if END reaches the last index? We are doing mod with addition. So, END goes back to index 0([0-(n-1)] will always be the range for END).

Second, let us see the pop function. In Queue pop removes and returns the front element. So, START needs to be modified. The general approach is to copy the current element pointed by START and increase the START variable to the next index.

**Corner case 3:** What if the Queue is empty? That’s why we are checking the START variable. If it is -1, then the queue is empty, we just exit.

**Corner case 4:** What if START goes out of bound? As done for END, mod addition comes to the rescue.

**Corner case 5:** What happens after popping the last element? We check this state with the currSize variable. Queue returns to the initial state, both START and END are set to -1.

Third, let’s see the top function. It behaves more like a pop function. We need to return the element pointed by the START variable. Since we are not actually removing any element, it’s fine to ignore corner cases 4 and 5.

That’s all about the Queue class implementation. In the main function, we just initialize the Queue class to check all corner cases.

**Code:**

class Queue {

private int arr[];

private int start, end, currSize, maxSize;

public Queue() {

arr = new int[16];

start = -1;

end = -1;

currSize = 0;

}

public Queue(int maxSize) {

this.maxSize = maxSize;

arr = new int[maxSize];

start = -1;

end = -1;

currSize = 0;

}

public void push(int newElement) {

if (currSize == maxSize) {

System.out.println("Queue is full\nExiting...");

System.exit(1);

}

if (end == -1) {

start = 0;

end = 0;

} else

end = (end + 1) % maxSize;

arr[end] = newElement;

System.out.println("The element pushed is " + newElement);

currSize++;

}

public int pop() {

if (start == -1) {

System.out.println("Queue Empty\nExiting...");

System.exit(1);

}

int popped = arr[start];

if (currSize == 1) {

start = -1;

end = -1;

} else

start = (start + 1) % maxSize;

currSize--;

return popped;

}

public int top() {

if (start == -1) {

System.out.println("Queue is Empty");

System.exit(1);

}

return arr[start];

}

public int size() {

return currSize;

}

}

public class TUF {

public static void main(String args[]) {

Queue q = new Queue(6);

q.push(4);

q.push(14);

q.push(24);

q.push(34);

System.out.println("The peek of the queue before deleting any element " + q.top());

System.out.println("The size of the queue before deletion " + q.size());

System.out.println("The first element to be deleted " + q.pop());

System.out.println("The peek of the queue after deleting an element " + q.top());

System.out.println("The size of the queue after deleting an element " + q.size());

}

}

**Output:**

The element pushed is 4  
The element pushed is 14  
The element pushed is 24  
The element pushed is 34  
The peek of the queue before deleting any element 4  
The size of the queue before deletion 4  
The first element to be deleted 4  
The peek of the queue after deleting an element 14  
The size of the queue after deleting an element 3

**Time Complexity:**

pop function: O(1)

push function: O(1)

top function: O(1)

size function: O(1)

**Space Complexity:**

Whole Queue: O(n)

**Check for Balanced Parentheses**

**Problem Statement:**Check Balanced Parentheses. Given string str containing just the characters ‘(‘, ‘)’, ‘{‘, ‘}’, ‘[‘ and ‘]’, check if the input string is valid and return true if the string is balanced otherwise return false.

**Note**: string str is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

**Example 1:**

**Input:** str = “( )[ { } ( ) ]”

**Output:** True

**Explanation**: As every open bracket has its corresponding

close bracket. Match parentheses are in correct order

hence they are balanced.

**Example 2:**

**Input:** str = “[ ( )”

**Output:** False

**Explanation**: As ‘[‘ does not have ‘]’ hence it is

not valid and will return false.

**Solution**

***Disclaimer***: *Don’t jump directly to the solution, try it out yourself first.*

**Intuition:** We have to keep track of previous as well as most recent opening brackets and also keep in mind the sequence, as after opening of the bracket there should be opposite pairs of brackets. Also handle the corner cases like [ ) ( ] where closing bracket occurs first and opening bracket after it, which is an invalid sequence, as well as [ ( ] ) where the most recent opening didn’t get its opposite pair hence it will also not be valid.

So we have to use a data structure that will keep track of first in and last out, hence we will use the **stack**.

**Approach:**

* Whenever we get the opening bracket we will push it into the stack. I.e ‘{‘, ’[’, ’(‘.
* Whenever we get the closing bracket we will check if the stack is non-empty or not.
* If the stack is empty we will return false, else if it is nonempty then we will check if the topmost element of the stack is the opposite pair of the closing bracket or not.
* If it is not the opposite pair of the closing bracket then return false, else move ahead.
* After we move out of the string the stack has to be empty if it is non-empty then return it as invalid else it is a valid string.

**Code:**

import java.util.\*;

class TUF {

public static boolean isValid(String s) {

Stack<Character> st = new Stack<Character>();

for (char it : s.toCharArray()) {

if (it == '(' || it == '[' || it == '{')

st.push(it);

else {

if(st.isEmpty()) return false;

char ch = st.pop();

if((it == ')' && ch == '(') || (it == ']' && ch == '[') || (it == '}' && ch == '{')) continue;

else return false;

}

}

return st.isEmpty();

}

public static void main (String[] args) {

String s="()[{}()]";

if(isValid(s)==true)

System.out.println("True");

else

System.out.println("False");

}

}

**Output:** True

**Time Complexity:**O(N)

**Space Complexity:**O(N)

**Next Greater Element Using Stack**

**Problem Statement:** Given a circular integer array **A**, return the next greater element for every element in A. The next greater element for an element x is the first element greater than x that we come across while traversing the array in a clockwise manner. If it doesn’t exist, return -1 for this element.

**Examples**:

**Example 1:**

**Input:** N = 11, A[] = {3,10,4,2,1,2,6,1,7,2,9}

**Output:** 10,-1,6,6,2,6,7,7,9,9,10

**Explanation:** For the first element in A ,i.e, 3, the greater element which comes next to it while traversing and is closest to it is 10. Hence,10 is present on index 0 in the resultant array. Now for the second element,i.e, 10, there is no greater number and hence -1 is it’s next greater element (NGE). Similarly, we got the NGEs for all other elements present in A.

**Example 2:**

**Input:** N = 6, A[] = {5,7,1,7,6,0}

**Output:** 7,-1,7,-1,7,5

**Solution**

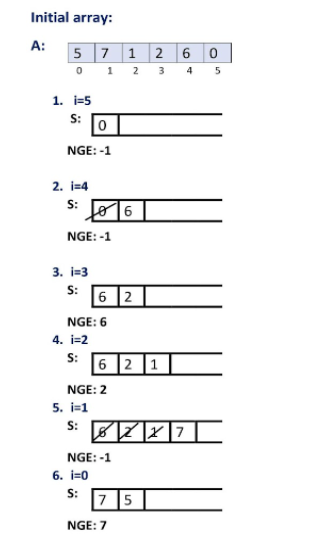
***Disclaimer***: *Don’t jump directly to the solution, try it out yourself first.*

**Approach:**

This problem can be solved easily and efficiently by using the stack data structure as it is based on the Last in First out (LIFO) principle.

To make it a bit easier let’s first try to solve without considering the array as circular. To find the next greater element we start traversing the given array from the right. As for the rightmost element, there is no other element at its right. Hence, we assign -1 at its index in the resultant array. Since this can be the next greater element (NGE) for some other element, we push it in the stack S. We keep checking for other elements. Let’s say we are checking for an element at index i. We keep popping from the stack until the element at the top of the stack is smaller than A[i]. The main intuition behind popping them is that these elements can never be the NGE for any element present at the left of A[i] because A[i] is greater than these elements. Now, if the top element of S is greater than A[i] then this is NGE of A[i] and we will assign it to res[i], where res is the resultant array. If the stack becomes empty then it implies that no element at the right of A[i] is greater than it and we assign -1. At last, we push A[i] in S.

**Dry run:** Let’s apply this algorithm for A[] = {5,7,1,2,6,0}:



So, the resultant array is {7,-1,2,6,-1,-1}. Remember that we have considered the array to be non-circular. For a circular array, the resultant array should be {7,-1,2,6,7,5}.

Now we need to make this algorithm work for a circular array. The only difference between a circular and non-circular array is that while searching for the next greater element in a non-circular array we don’t consider the elements left to the concerned element. This can be easily done by inserting the elements of the array A at the end of A, thus making its size double. But we actually don’t require any extra space. We can just traverse the array twice. We actually run a loop 2\*N times, where N is the size of the given array.

**Code:**

import java.io.\*;

import java.util.\*;

class TUF {

public static int[] nextGreaterElements(int[] nums) {

int n = nums.length;

int nge[] = new int[n];

Stack < Integer > st = new Stack < > ();

for (int i = 2 \* n - 1; i >= 0; i--) {

while (st.isEmpty() == false && st.peek() <= nums[i % n]) {

st.pop();

}

if (i < n) {

if (st.isEmpty() == false) nge[i] = st.peek();

else nge[i] = -1;

}

st.push(nums[i % n]);

}

return nge;

}

public static void main(String args[]) {

int arr[]={5,7,1,2,6,0};

int arr2[] = nextGreaterElements(arr);

System.out.println("The next greater elements are ");

for (int i = 0; i < arr2.length; i++) {

System.out.print(arr2[i] + " ");

}

}

}

**Output:**

The next greater elements are  
7 -1 2 6 7 5

**Time Complexity: O(N)**

**Space Complexity: O(N)**

# Implement Min Stack : O(2N) and O(N) Space Complexity

**Problem Statement:** Implement Min Stack | O(2N) and O(N) Space Complexity. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

**Examples**:

**Input Format**:["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]

[

[ ], [-2], [0], [-3], [ ], [ ], [ ], [ ]

]

**Result:** [null, null, null, null, -3, null, 0, -2]

**Explanation:**

stack < long long > st

st.push(-2); Push element in stack

st.push(0); Push element in stack

st.push(-3); Push element in stack

st.getMin(); Get minimum element fromstack

st.pop(); Pop the topmost element

st.top(); Top element is 0

st.getMin(); Minimum element is -2

**Solution:**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1:**Using pairs to store the value and minimum element till now.

**Approach**: The first element in the pair will store the value and the second element will store the minimum element till now.

When the first push operation comes in we will push the value and store it as minimum itself in the pair.

In the second push operation, we will check if the top element’s minimum is less than the new value. If it is then we will push the value with minimum as the previous top’s minimum. To get the getMin element to take the top’s second element.

**Code:**

class Pair {

int x, y;

Pair(int x, int y) {

this.x = x;

this.y = y;

}

}

class MinStack {

Stack < Pair > st;

public MinStack() {

st = new Stack < > ();

}

public void push(int x) {

int min;

if (st.isEmpty()) {

min = x;

} else {

min = Math.min(st.peek().y, x);

}

st.push(new Pair(x, min));

}

public void pop() {

st.pop();

}

public int top() {

return st.peek().x;

}

public int getMin() {

return st.peek().y;

}

}

**Time Complexity: O(1)**

**Space Complexity: O(2N)**

**Solution 2:**

**Approach:**

Let’s take a variable that stores the minimum number. So whenever a push operation comes in just take that number put it in the stack and update the variable to the number.

**Push operation:**

Now if there is a push operation just check whether that number is less than the min number. If it is smaller than min we will push a modified value which is a push(2 \* Val – min) into the stack and will update min to the value of the original number. If it’s not then we will just push it as it is.

**getMin() operation:**

We will just return the value of min.

**Top operation:**

While returning the top value we know that it is a modified value. We will check if the top value is lesser than min, If it is then we will return the min as the top value.

**Pop operation:**

While making pop we will check if the top value is lesser than min, If it is then we must update our min to its previous value. In order to do that min = (2 \* min) – (modified value) and we will pop the element.

**Code:**

class MinStack {

Stack < Long > st = new Stack < Long > ();

Long mini;

/\*\* initialize your data structure here. \*/

public MinStack() {

mini = Long.MAX\_VALUE;

}

public void push(int value) {

Long val = Long.valueOf(value);

if (st.isEmpty()) {

mini = val;

st.push(val);

} else {

if (val < mini) {

st.push(2 \* val - mini);

mini = val;

} else {

st.push(val);

}

}

}

public void pop() {

if (st.isEmpty()) return;

Long val = st.pop();

if (val < mini) {

mini = 2 \* mini - val;

}

}

public int top() {

Long val = st.peek();

if (val < mini) {

return mini.intValue();

}

return val.intValue();

}

public int getMin() {

return mini.intValue();

}

}

**Time Complexity: O(1)**

**Space Complexity: O(N)**