Implement Stack using Array

Stack is a **linear data structure** which follows **LIFO** principle. In this article, we will learn how to implement Stack using Arrays. In Array-based approach, all stack-related operations are executed using arrays. Let's see how we can implement each operation on the stack utilizing the Array Data Structure.

Implement Stack using Array:

*To implement a stack using an array, initialize an array and treat its end as the stack's top. Implement **push** (add to end), **pop** (remove from end), and **peek** (check end) operations, handling cases for an **empty** or f**ull stack**.*
Recommended Problem

Step-by-step approach:

1. **Initialize an array** to represent the stack.
2. Use the **end of the array** to represent the **top of the stack**.
3. Implement **push** (add to end), **pop** (remove from the end), and **peek** (check end) operations, ensuring to handle empty and full stack conditions.

Implement Stack Operations using Array:

Here are the following operations of implement stack using array:

Push Operation in Stack:

Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**

Algorithm for Push Operation:

- *Before pushing the element to the stack, we check if the stack is **full** .*
- *If the stack is full **(top == capacity-1)** , then **Stack Overflows** and we cannot insert the element to the stack.*
- *Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at **top position** .*
- *The elements can be pushed into the stack till we reach the **capacity** of the stack.*

Pop Operation in Stack:

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

Algorithm for Pop Operation:

- *Before popping the element from the stack, we check if the stack is **empty** .*

- *If the stack is empty (top == -1), then **Stack Underflows** and we cannot remove any element from the stack.*
- *Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)** and return the stored top value.*

Top or Peek Operation in Stack:

Returns the top element of the stack.

Algorithm for Top Operation:

- *Before returning the top element from the stack, we check if the stack is empty.*
- *If the stack is empty (top == -1), we simply print "Stack is empty".*
- *Otherwise, we return the element stored at **index = top** .*

isEmpty Operation in Stack:

Returns true if the stack is empty, else false.

Algorithm for isEmpty Operation :

- *Check for the value of **top** in stack.*
- *If **(top == -1)** , then the stack is **empty** so return **true** .*
- *Otherwise, the stack is not empty so return **false** .*

isFull Operation in Stack :

Returns true if the stack is full, else false.

Algorithm for isFull Operation:

- *Check for the value of **top** in stack.*
- *If **(top == capacity-1),** then the stack is **full** so return **true** .*
- *Otherwise, the stack is not full so return **false.***

Below is the implementation of the above approach:

C++CJavaPython3C#JavaScript

```
/* C++ program to implement basic stack
operations */
#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
```

```
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    class Stack s;
```

```
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";

    //print top element of stack after popping
    cout << "Top element is : " << s.peek() << endl;

    //print all elements in stack :
    cout <<"Elements present in stack : ";
    while(!s.isEmpty())
    {
        // print top element in stack
        cout << s.peek() <<" ";
        // remove top element in stack
        s.pop();
    }

    return 0;
}
```

Output

```
10 pushed into stack

20 pushed into stack

30 pushed into stack

30 Popped from stack

Top element is : 20

Elements present in stack : 20 10
```

Complexity Analysis:

- **Time Complexity**:
    - `push`: O(1)
    - `pop`: O(1)
    - `peek`: O(1)
    - `is_empty`: O(1)
    - is_full: O(1)
- **Auxiliary Space**: O(n), where n is the number of items in the stack.

Advantages of Array Implementation:

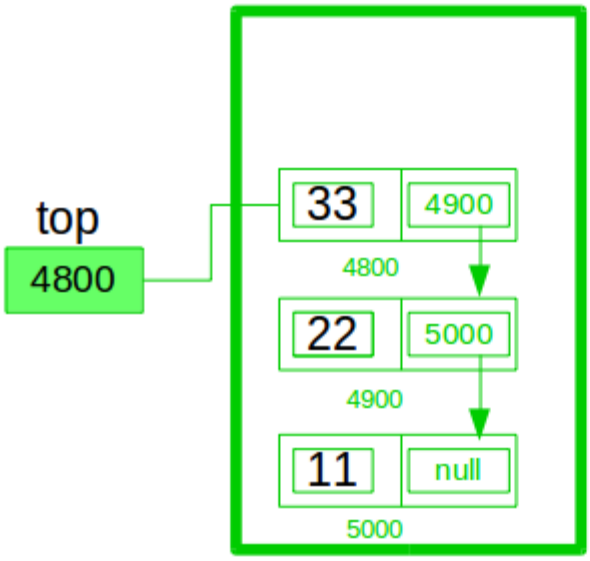
- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of Array Implementation:

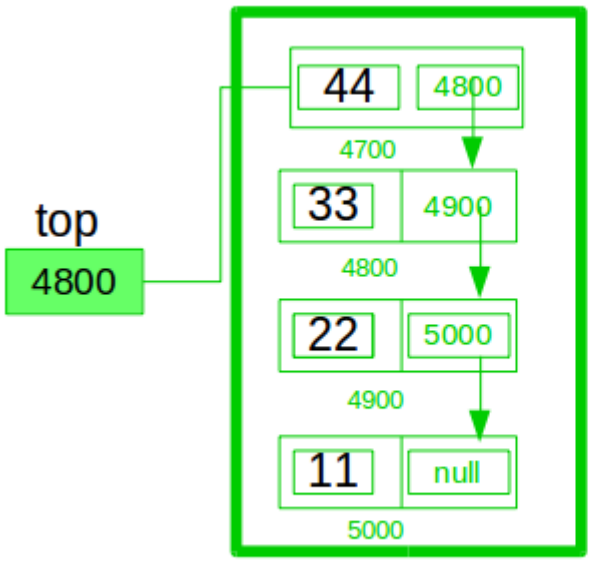
- It is not dynamic i.e., it doesn't grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
- The total size of the stack must be defined beforehand.

Implement a stack using singly linked list

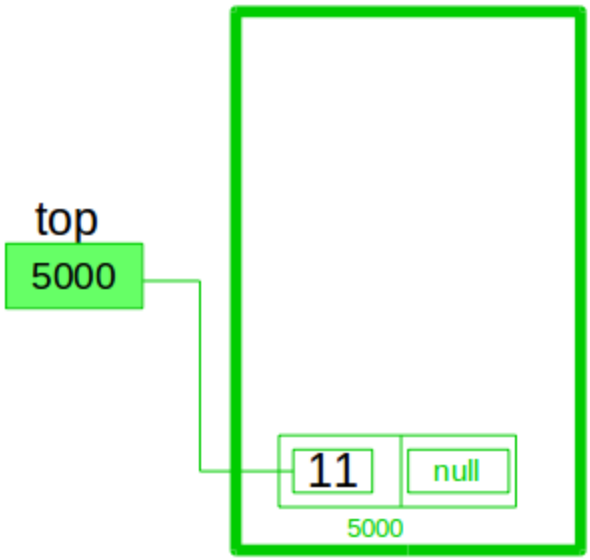
- To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.
  So we need to follow a simple rule in the implementation of a stack which is **last in first out** and all the operations can be performed with the help of a top variable. Let us learn how to perform **Pop, Push, Peek, and Display** operations in the following article:



Initial Stack Having Three element
And top have address 4800



First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top




```
Pop three  element
      temp = top;
       top = top->link;
       temp->link = NULL;
       free(temp);
```

In the stack Implementation, a stack contains a top pointer. which is the “head” of the stack where pushing and popping items happens at the head of the list. The first node has a null in the link field and second node-link has the first node address in the link field and so on and the last node address is in the “top” pointer.
The main advantage of using a linked list over arrays is that it is possible to implement a stack that can shrink or grow as much as needed. Using an array will put a restriction on the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated. so overflow is not possible.

Stack Operations:

- **push():** Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
- **pop():** Return the top element of the Stack i.e simply delete the first element from the linked list.
- **peek():** Return the top element.
- **display():** Print all elements in Stack.

Push Operation:

- *Initialise a node*
- *Update the value of that node by data i.e.* ***node->data = data***
- *Now link this node to the top of the linked list*
- *And update top pointer to the current node*

Recommended Problem

[Implement Stack using Linked List]

Pop Operation:

- *First Check whether there is any node present in the linked list or not, if not then return*
- *Otherwise make pointer let say **temp** to the top node and move forward the top node by 1 step*
- *Now free this temp node*

Peek Operation:

- *Check if there is any node present or not, if not then return.*
- *Otherwise return the value of top node of the linked list*

Display Operation:

- *Take a **temp** node and initialize it with top pointer*
- *Now start traversing temp till it encounters NULL*
- *Simultaneously print the value of the temp node*

Below is the implementation of the above operations
C++CJavaPythonC#JavaScript

```cpp
// C++ program to implement a stack using singly linked list
#include <bits/stdc++.h>
using namespace std;

// Class representing a node in the linked list
class Node {
public:
    int data;
    Node* next;
    Node(int new_data) {
        this->data = new_data;
        this->next = nullptr;
    }
};

// Class to implement stack using a singly linked list
class Stack {

    // head of the linked list
    Node* head;

public:
    // Constructor to initialize the stack
    Stack() { this->head = nullptr; }

    // Function to check if the stack is empty
```

```
    bool isEmpty() {

        // If head is nullptr, the stack is empty
        return head == nullptr;
    }

    // Function to push an element onto the stack
    void push(int new_data) {

        // Create a new node with given data
        Node* new_node = new Node(new_data);

        // Check if memory allocation for the new node
        // failed
        if (!new_node) {
            cout << "\nStack Overflow";
        }

        // Link the new node to the current top node
        new_node->next = head;

        // Update the top to the new node
        head = new_node;
    }

    // Function to remove the top element from the stack
    void pop() {

        // Check for stack underflow
        if (this->isEmpty()) {
            cout << "\nStack Underflow" << endl;
        }
        else {
            // Assign the current top to a temporary
            // variable
            Node* temp = head;

            // Update the top to the next node
            head = head->next;

            // Deallocate the memory of the old top node
            delete temp;
        }
    }

    // Function to return the top element of the stack
    int peek() {
```

```
        // If stack is not empty, return the top element
        if (!isEmpty())
            return head->data;
        else {
            cout << "\nStack is empty";
            return INT_MIN;
        }
    }
};

// Driver program to test the stack implementation
int main() {
    // Creating a stack
    Stack st;

    // Push elements onto the stack
    st.push(11);
    st.push(22);
    st.push(33);
    st.push(44);

    // Print top element of the stack
    cout << "Top element is " << st.peek() << endl;

    // removing two elememements from the top
      cout << "Removing two elements..." << endl;
    st.pop();
    st.pop();

    // Print top element of the stack
    cout << "Top element is " << st.peek() << endl;

    return 0;
}
```

Output

```
Top element is 44

Top element is 22
```

Time Complexity: O(1), for all push(), pop(), and peek(), as we are not performing any kind of traversal over the list. We perform all the operations through the current pointer only.
Auxiliary Space: O(N), where N is the size of the stack
In this implementation, we define a Node class that represents a node in the linked list, and a Stack class that uses this node class to implement the stack. The head

attribute of the Stack class points to the top of the stack (i.e., the first node in the linked list).

To push an item onto the stack, we create a new node with the given item and set its next pointer to the current head of the stack. We then set the head of the stack to the new node, effectively making it the new top of the stack.

To pop an item from the stack, we simply remove the first node from the linked list by setting the head of the stack to the next node in the list (i.e., the node pointed to by the next pointer of the current head). We return the data stored in the original head node, which is the item that was removed from the top of the stack.

Benefits of implementing a stack using a singly linked list include:

Dynamic memory allocation: The size of the stack can be increased or decreased dynamically by adding or removing nodes from the linked list, without the need to allocate a fixed amount of memory for the stack upfront.

Efficient memory usage: Since nodes in a singly linked list only have a next pointer and not a prev pointer, they use less memory than nodes in a doubly linked list.

Easy implementation: Implementing a stack using a singly linked list is straightforward and can be done using just a few lines of code.

Versatile: Singly linked lists can be used to implement other data structures such as queues, linked lists, and trees.

In summary, implementing a stack using a singly linked list is a simple and efficient way to create a dynamic stack data structure in Python.

Real time examples of stack:

Stacks are used in various real-world scenarios where a last-in, first-out (LIFO) data structure is required. Here are some examples of real-time applications of stacks:

Function call stack: When a function is called in a program, the return address and all the function parameters are pushed onto the function call stack. The stack allows the function to execute and return to the caller function in the reverse order in which they were called.

Undo/Redo operations: In many applications, such as text editors, image editors, or web browsers, the undo and redo functionalities are implemented using a stack. Every time an action is performed, it is pushed onto the stack. When the user wants to undo the last action, the top element of the stack is popped and the action is reversed.

Browser history: Web browsers use stacks to keep track of the pages visited by the user. Every time a new page is visited, its URL is pushed onto the stack. When the user clicks the “Back” button, the last visited URL is popped from the stack and the user is directed to the previous page.

Expression evaluation: Stacks are used in compilers and interpreters to evaluate expressions. When an expression is parsed, it is converted into postfix notation and pushed onto a stack. The postfix expression is then evaluated using the stack.

Call stack in recursion: When a recursive function is called, its call is pushed onto the stack. The function executes and calls itself, and each subsequent call is pushed

onto the stack. When the recursion ends, the stack is popped, and the program returns to the previous function call.
In summary, stacks are widely used in many applications where LIFO functionality is required, such as function calls, undo/redo operations, browser history, expression evaluation, and recursive function calls.

Balanced Parenthesis

Given an expression string **exp**, write a program to examine whether the pairs and the orders of "{", "}", "(", ")", "[", "]" are correct in the given expression.

Example:

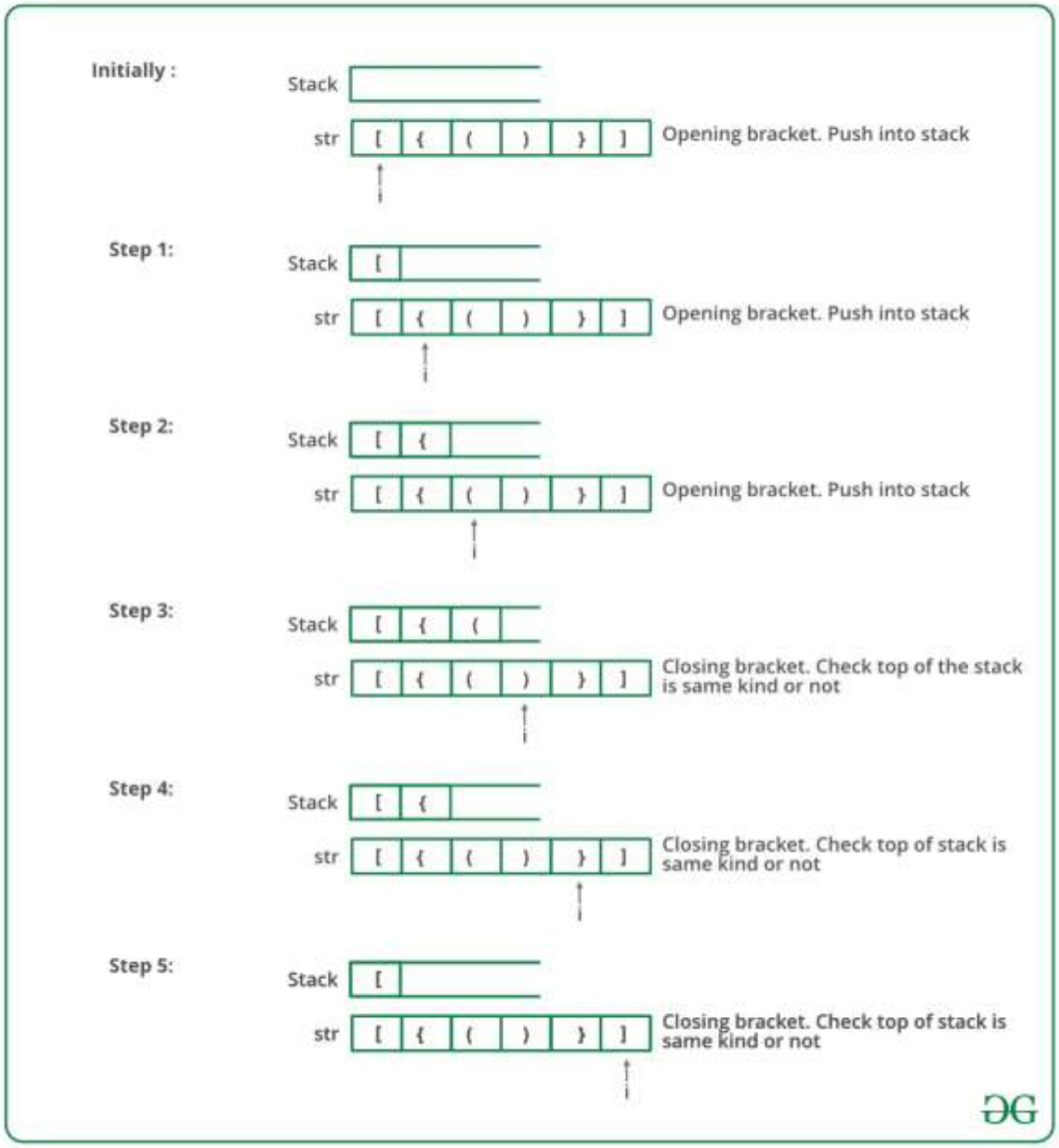
***Input**: exp = "[( )] { } { [( ) ( )] ( ) }"*
***Output**: Balanced*
***Explanation:** all the brackets are well-formed*

***Input**: exp = "[( ])"*
***Output**: Not Balanced*
***Explanation: 1** and **4** brackets are not balanced because there is a closing '] ' before the closing '('*

Check for Balanced Bracket expression using Stack:

The idea is to put all the opening brackets in the stack. Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature. If this holds then pop the stack and continue the iteration, in the end if the stack is empty, it means all brackets are well-formed . Otherwise, they are not balanced.

Illustration:
Below is the illustration of the above approach.



Follow the steps mentioned below to implement the idea:

- Declare a character stack (say **temp**).
- Now traverse the string exp.
  - If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.
  - If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then pop from stack and if the popped character is the matching starting bracket then fine.
  - Else brackets are **Not Balanced**.
- After complete traversal, if there is some starting bracket left in stack then **Not balanced**, else **Balanced**.

Below is the implementation of the above approach:

C++Java

```
// C++ program to check for balanced brackets.

#include <bits/stdc++.h>
using namespace std;

// Function to check if brackets are balanced
bool areBracketsBalanced(string expr)
{
	// Declare a stack to hold the previous brackets.
	stack<char> temp;
	for (int i = 0; i < expr.length(); i++) {
		if (temp.empty()) {

			// If the stack is empty
			// just push the current bracket
			temp.push(expr[i]);
		}
		else if ((temp.top() == '(' && expr[i] == ')')
				|| (temp.top() == '{' && expr[i] == '}')
				|| (temp.top() == '[' && expr[i] == ']')) {

			// If we found any complete pair of bracket
			// then pop
			temp.pop();
		}
		else {
			temp.push(expr[i]);
		}
	}
	if (temp.empty()) {

		// If stack is empty return true
		return true;
	}
	return false;
}

// Driver code
int main()
{
	string expr = "{()}[]";
```

```
	// Function call
	if (areBracketsBalanced(expr))
		cout << "Balanced";
	else
		cout << "Not Balanced";
	return 0;
}
```

Output
```
Balanced
```
Time Complexity: O(N), Iteration over the string of size N one time.

Auxiliary Space: O(N) for stack.

Evaluation of Postfix Expression

Given a postfix expression, the task is to evaluate the postfix expression.

Postfix expression: *The expression of the form "a b operator" (ab+) i.e., when a pair of operands is followed by an operator.*

Examples:
Input*: str = "2 3 1 * + 9 -"*
Output*: -4*
Explanation: *If the expression is converted into an infix expression, it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.*
Input*: str = "100 200 + 2 / 5 * 7 +"*
Output*: 757*

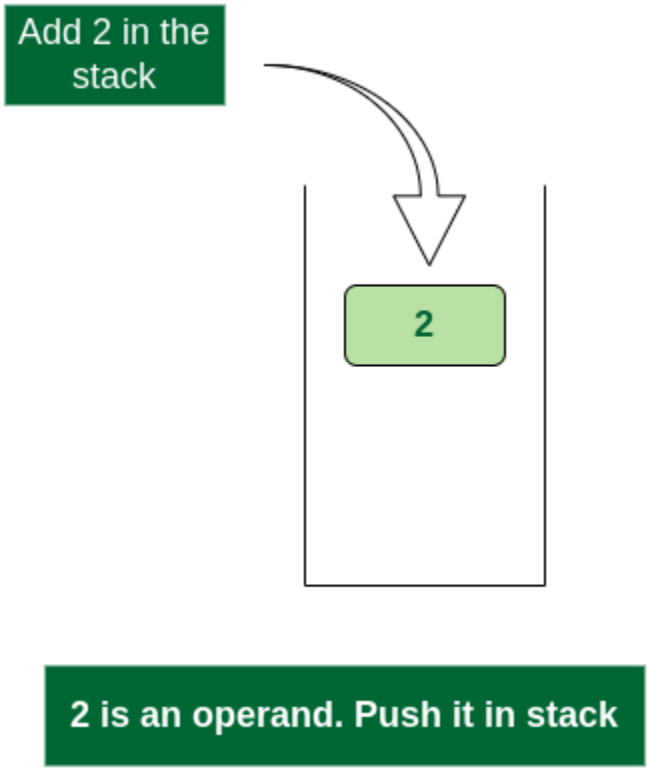
Evaluation of Postfix Expression using Stack:

To evaluate a postfix expression we can use a stack.
Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

Illustration:
Follow the below illustration for a better understanding:

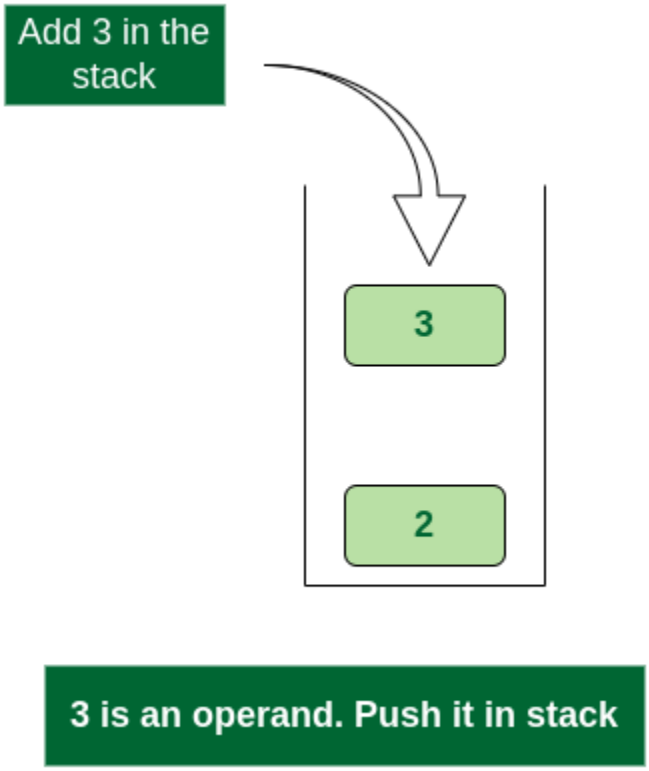
Consider the expression: exp = ***"2 3 1 * + 9 -"***

- *Scan* ***2****, it's a number, So push it into stack. Stack contains '2'.*



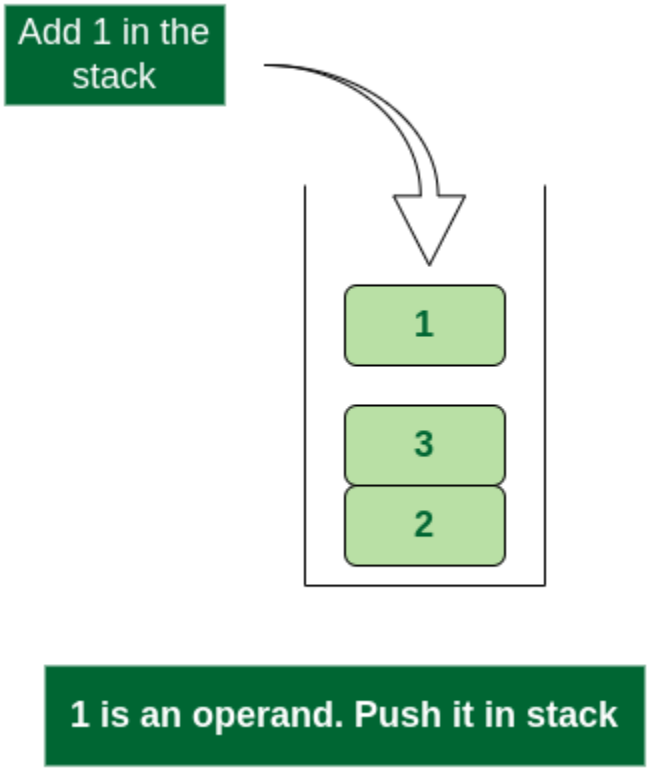
Push 2 into stack

- *Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top)*



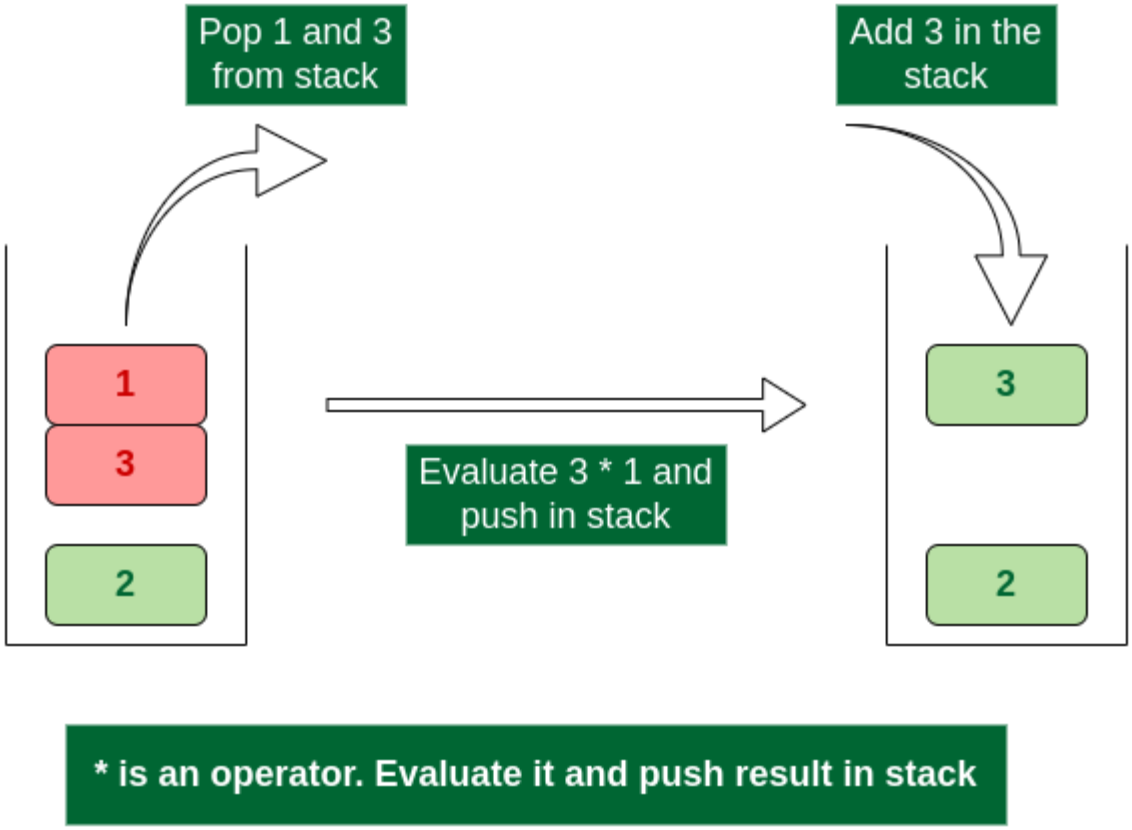
Push 3 into stack

- *Scan 1, again a number, push it to stack, stack now contains ‘2 3 1’*



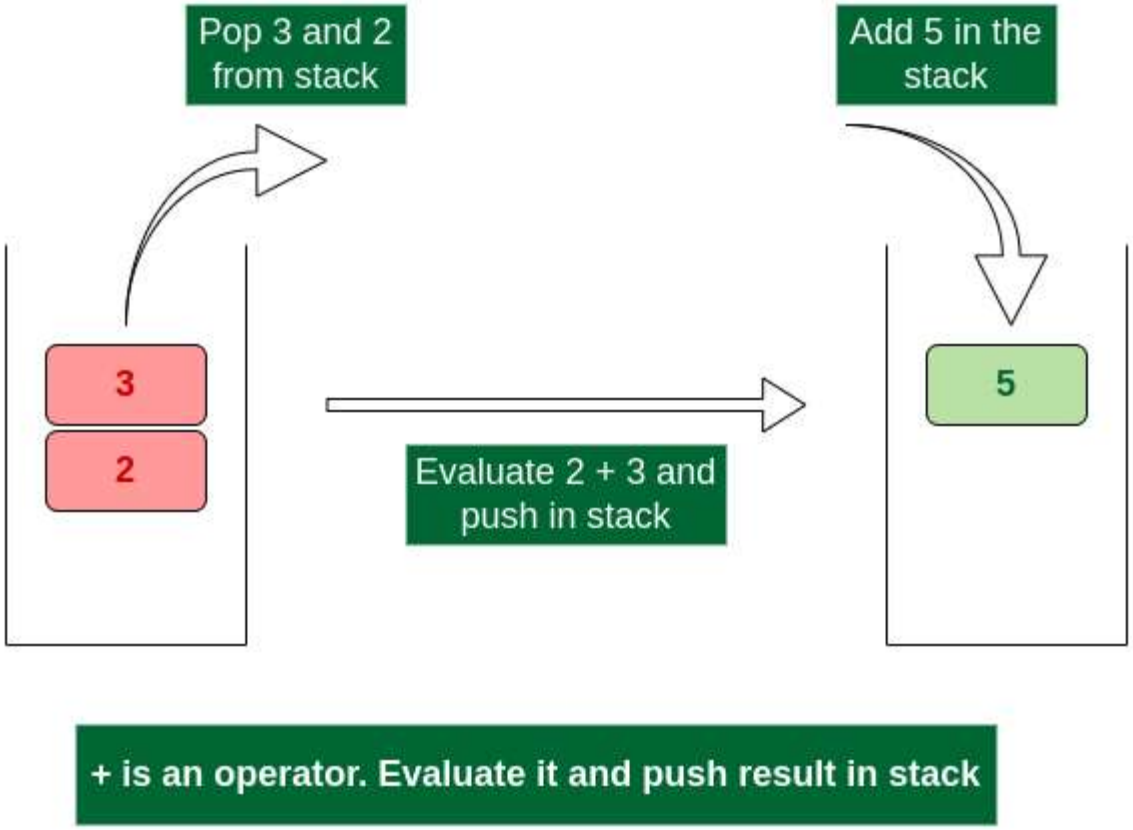
Push 1 into stack

- *Scan *, it's an operator. Pop two operands from stack, apply the * operator on operands. We get 3*1 which results in 3. We push the result 3 to stack. The stack now becomes '2 3'.*



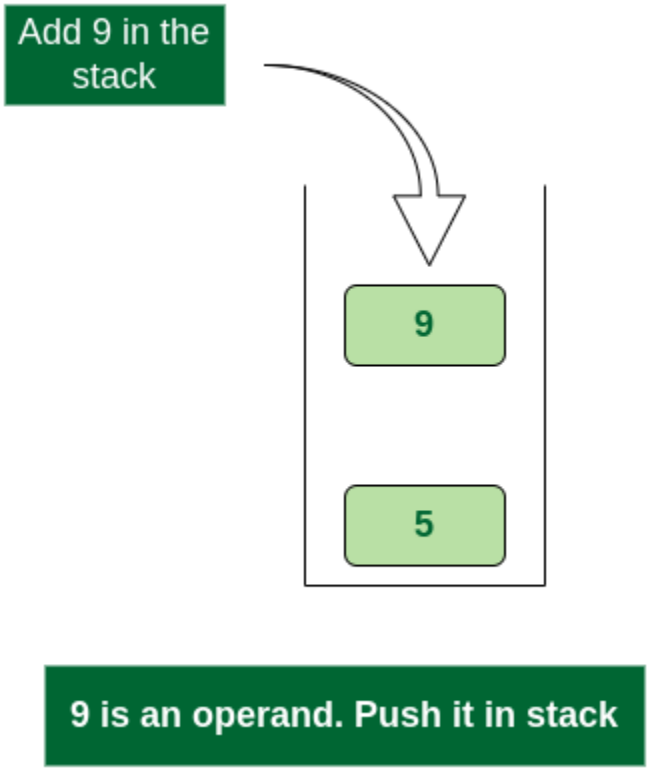
*Evaluate * operator and push result in stack*

- *Scan +, it's an operator. Pop two operands from stack, apply the + operator on operands. We get 3 + 2 which results in 5. We push the result 5 to stack. The stack now becomes '5'.*



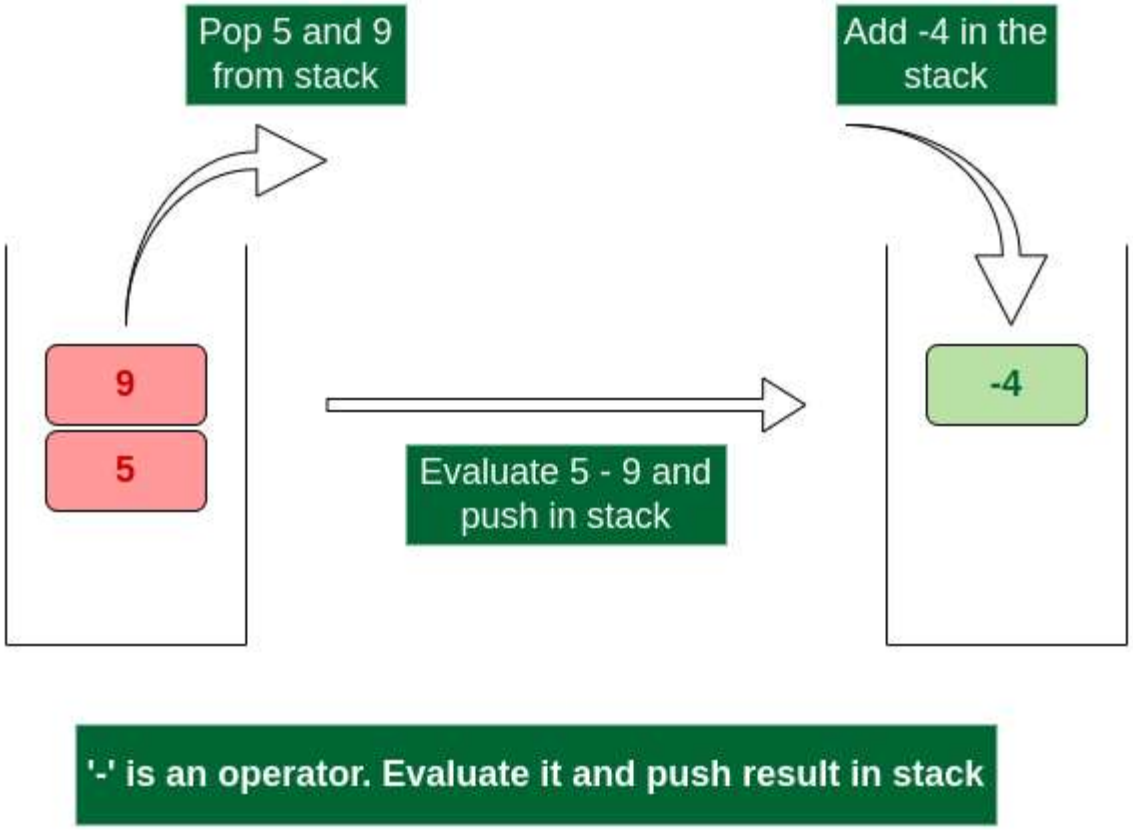
Evaluate + operator and push result in stack

- *Scan 9, it's a number. So we push it to the stack. The stack now becomes '5 9'.*



Push 9 into stack

- *Scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result -4 to the stack. The stack now becomes '-4'.*



Evaluate '-' operator and push result in stack

- *There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).*

*So the result becomes **-4**.*

Follow the steps mentioned below to evaluate postfix expression using stack:

- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
    - If the element is a number, push it into the stack.
    - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
- When the expression is ended, the number in the stack is the final answer.

Below is the implementation of the above approach:

- C

```
// C++ program to evaluate value of a postfix expression
#include <bits/stdc++.h>
using namespace std;

// The main function that returns value
// of a given postfix expression
int evaluatePostfix(string exp)
{
    // Create a stack of capacity equal to expression size
    stack<int> st;

    // Scan all characters one by one
    for (int i = 0; i < exp.size(); ++i) {

        // If the scanned character is an operand
        // (number here), push it to the stack.
        if (isdigit(exp[i]))
            st.push(exp[i] - '0');

        // If the scanned character is an operator,
        // pop two elements from stack apply the operator
        else {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            switch (exp[i]) {
            case '+':
                st.push(val2 + val1);
                break;
            case '-':
                st.push(val2 - val1);
                break;
            case '*':
                st.push(val2 * val1);
                break;
            case '/':
                st.push(val2 / val1);
                break;
            }
        }
    }
    return st.top();
}

// Driver code
int main()
{
    string exp = "231*+9-";

    // Function call
```

```
    cout << "postfix evaluation: " << evaluatePostfix(exp);
    return 0;
}
```

Output

```
postfix evaluation: -4
```

Time Complexity: O(N)
Auxiliary Space: O(N)

There are the following limitations of the above implementation.

- It supports only **4** binary operators **'+', '*', '-'** and **'/'**. It can be extended for more operators by adding more switch cases.
- The allowed operands are only single-digit operands.

Postfix evaluation for multi-digit numbers:

*The above program can be extended for multiple digits by adding a separator-like space between all elements **(operators and operands)** of the given expression.*

Below given is the extended program which allows operands to have multiple digits.

```
// C program to evaluate value of a postfix
// expression having multiple digit operands
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Stack type
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack
        = (struct Stack*)malloc(sizeof(struct Stack));

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;
```

```
    stack->array
        = (int*)malloc(stack->capacity * sizeof(int));

    if (!stack->array)
        return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

int peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

int pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '$';
}

void push(struct Stack* stack, int op)
{
    stack->array[++stack->top] = op;
}

// The main function that returns value
// of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack)
        return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i) {
        // if the character is blank space then continue
        if (exp[i] == ' ')
            continue;

        // If the scanned character is an
        // operand (number here),extract the full number
```

```
        // Push it to the stack.
        else if (isdigit(exp[i])) {
            int num = 0;

            // extract full number
            while (isdigit(exp[i])) {
                num = num * 10 + (int)(exp[i] - '0');
                i++;
            }
            i--;

            // push the element in the stack
            push(stack, num);
        }

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else {
            int val1 = pop(stack);
            int val2 = pop(stack);

            switch (exp[i]) {
            case '+':
                push(stack, val2 + val1);
                break;
            case '-':
                push(stack, val2 - val1);
                break;
            case '*':
                push(stack, val2 * val1);
                break;
            case '/':
                push(stack, val2 / val1);
                break;
            }
        }
    }
    return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "100 200 + 2 / 5 * 7 +";

    // Function call
    printf("%d", evaluatePostfix(exp));
    return 0;
}

// This code is contributed by Arnab Kundu
```

Output

757

Time Complexity: O(N)
Auxiliary Space: O(N)

Implement two Stacks in an Array

Create a data structure **twoStacks** that represent two stacks. Implementation of **twoStacks** should use only one array, i.e., both stacks should use the same array for storing elements.

Following functions must be supported by *twoStacks*.

- push1(int x) --> pushes x to first stack
- push2(int x) --> pushes x to second stack
- pop1() --> pops an element from first stack and return the popped element
- pop2() --> pops an element from second stack and return the popped element

Implementation of *two Stack* should be space efficient.

Implement two stacks in an array by Dividing the space into two halves:

The idea to implement two stacks is to divide the array into two halves and assign two halves to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.
Follow the steps below to solve the problem:

- To implement **push1():**
  - First, check whether the **top1** is greater than 0
    - If it is then add an element at the top1 index and decrement top1 by 1
    - Else return Stack Overflow
- To implement **push2():**
  - First, check whether **top2** is less than n - 1
    - If it is then add an element at the top2 index and increment the top2 by 1
    - Else return Stack Overflow

- To implement **pop1():**
    - First, check whether the **top1** is less than or equal to n / 2
        - If it is then increment the top1 by 1 and return that element.
        - Else return Stack Underflow
- To implement **pop2():**
    - First, check whether the **top2** is greater than or equal to (n + 1) / 2
        - If it is then decrement the top2 by 1 and return that element.
        - Else return Stack Underflow

Below is the implementation of the above approach.

C++Java

```
#include <bits/stdc++.h>
using namespace std;

class twoStacks {
    int* arr;
    int size;
    int top1, top2;

public:
    // Constructor
    twoStacks(int n)
    {
        size = n;
        arr = new int[n];
        top1 = n / 2 + 1;
        top2 = n / 2;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty
        // space for new element
        if (top1 > 0) {
            top1--;
            arr[top1] = x;
        }
        else {
            cout << "Stack Overflow"
                 << " By element : " << x << endl;
            return;
        }
    }

    // Method to push an element
    // x to stack2
```

```
    void push2(int x)
    {

        // There is at least one empty
        // space for new element
        if (top2 < size - 1) {
            top2++;
            arr[top2] = x;
        }
        else {
            cout << "Stack Overflow"
                 << " By element : " << x << endl;
            return;
        }
    }

    // Method to pop an element from first stack
    int pop1()
    {
        if (top1 <= size / 2) {
            int x = arr[top1];
            top1++;
            return x;
        }
        else {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }

    // Method to pop an element
    // from second stack
    int pop2()
    {
        if (top2 >= size / 2 + 1) {
            int x = arr[top2];
            top2--;
            return x;
        }
        else {
            cout << "Stack UnderFlow" << endl;
            exit(1);
        }
    }
};

/* Driver program to test twoStacks class */
int main()
{
```

```
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is "
         << ": " << ts.pop1() << endl;
    ts.push2(40);
    cout << "Popped element from stack2 is "
         << ": " << ts.pop2() << endl;
    return 0;
}
```

Output

```
Stack Overflow By element : 7
Popped element from stack1 is : 11
Stack Overflow By element : 40
Popped element from stack2 is : 15
```

Time Complexity:

- **Both Push operation:** O(1)
- **Both Pop operation:** O(1)

Auxiliary Space: O(N), Use of array to implement stack.

Problem in the above implementation:

The problem in the above implementation is that as we reserve half of the array for a stack and another half for the another stack. So, let if 1st half is full means first stack already have n/2 numbers of elements and 2nd half is not full means it doesn't have n/2 numbers of elements. So, if we look into the array, there are free spaces inside array(eg. in the next half) but we cannot push elements for stack 1(because first half is reserved for stack 1 and it's already full). It means this implementation show stack overflow although the array is not full. The solution for this answer is the below implementation.

Implement two stacks in an array by Starting from endpoints:

The idea is to start two stacks from two extreme corners of arr[].
Follow the steps below to solve the problem:

- Stack1 starts from the leftmost corner of the array, the first element in stack1 is pushed at index 0 of the array.
- Stack2 starts from the rightmost corner of the array, the first element in stack2 is pushed at index (n-1) of the array.
- Both stacks grow (or shrink) in opposite directions.
- To check for overflow, all we need to check is for availability of space between top elements of both stacks.
- To check for underflow, all we need to check is if the value of the top of the both stacks is between 0 to (n-1) or not.

Below is the implementation of above approach:

C++Java

```
#include <iostream>
#include <stdlib.h>

using namespace std;

class twoStacks {
    int* arr;
    int size;
    int top1, top2;

public:
    twoStacks(int n) // constructor
    {
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1) {
            top1++;
            arr[top1] = x;
        }
        else {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
```

```
    {
        // There is at least one empty
        // space for new element
        if (top1 < top2 - 1) {
            top2--;
            arr[top2] = x;
        }
        else {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to pop an element from first stack
    int pop1()
    {
        if (top1 >= 0) {
            int x = arr[top1];
            top1--;
            return x;
        }
        else {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }

    // Method to pop an element from second stack
    int pop2()
    {
        if (top2 < size) {
            int x = arr[top2];
            top2++;
            return x;
        }
        else {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }
};

/* Driver program to test twoStacks class */
int main()
{
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
```

```
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
    return 0;
}
```

Output

```
Popped element from stack1 is 11
Popped element from stack2 is 40
```

Time Complexity:

- **Both Push operation:** O(1)
- **Both Pop operation:** O(1)

Auxiliary Space: O(N), Use of the array to implement stack.

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element.

The **Next greater Element** for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider the next greater element as -1.

Example:

Input: arr[] = [4 , 5 , 2 , 25]
Output: 4 --> 5
5 --> 25
2 --> 25
25 --> -1
Explanation: *except* ***25*** *every element has an element greater than them present on the right side*

Input: arr[] = [13 , 7, 6 , 12]
Output: 13 --> -1
7 --> 12
6 --> 12

12 --> -1

***Explanation:** **13** and **12** don't have any element greater than them present on the right side*

Find Next Greater Element using Nested Loops:

The idea is to use two loops , The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by the outer loop. If a greater element is found then that element is printed as next, otherwise, -1 is printed.

Follow the steps mentioned below to implement the idea:

- Traverse The array from **index 0** to **end**.
- For each element start another loop from **index i+1** to **end.**
- If a greater element is found in the second loop then print it and **break** the loop, else print **-1.**

Below is the implementation of the above approach:

C++Java

```
// Simple C++ program to print
// next greater elements in a
// given array
#include <iostream>
using namespace std;

/* prints element and NGE pair
for all elements of arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i = 0; i < n; i++) {
        next = -1;
        for (j = i + 1; j < n; j++) {
            if (arr[i] < arr[j]) {
                next = arr[j];
                break;
            }
        }
        cout << arr[i] << " --> " << next << endl;
    }
}

// Driver Code
```

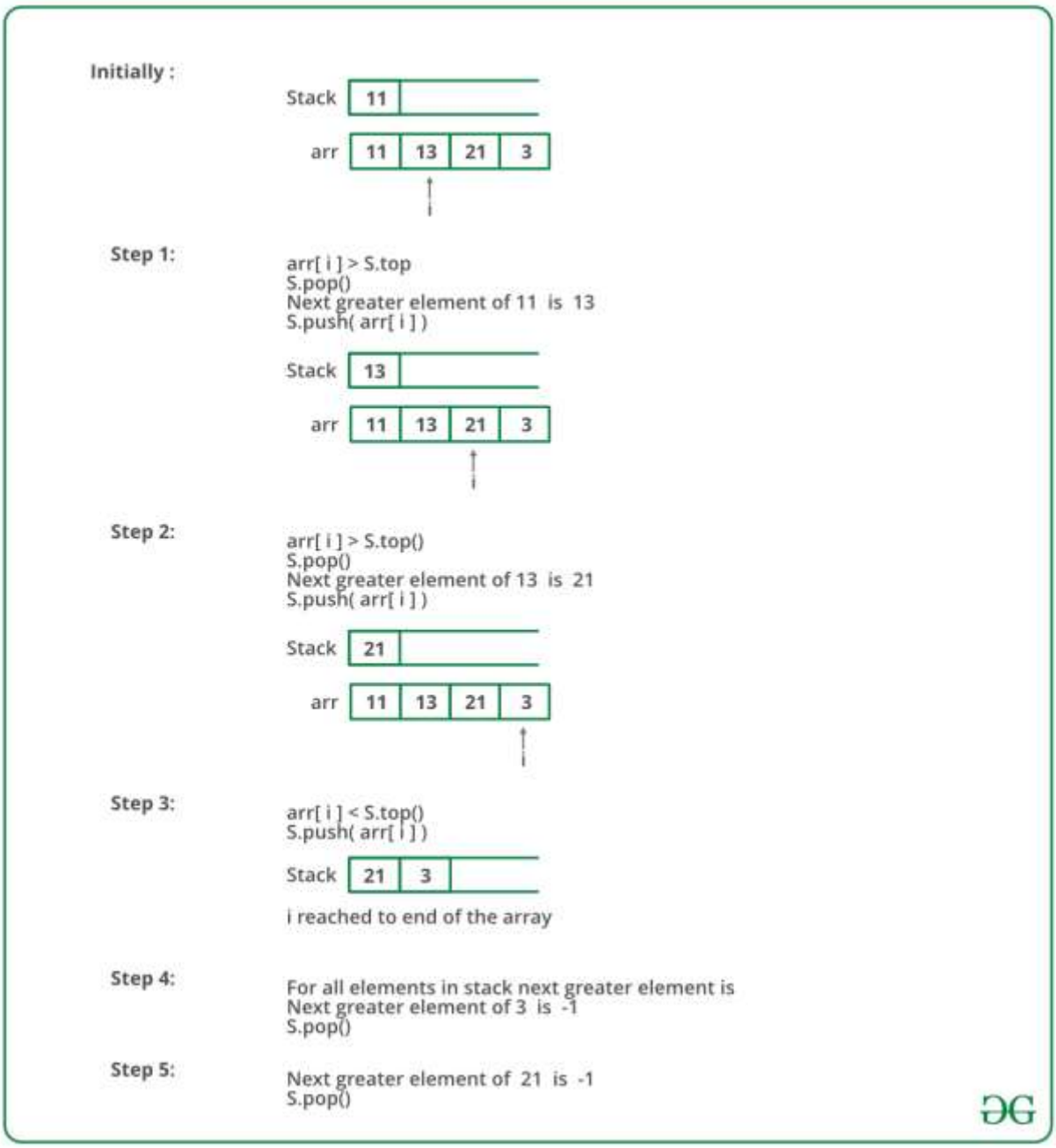
```
int main()
{
    int arr[] = { 11, 13, 21, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

Time Complexity: $O(N^2)$
Auxiliary Space: O(1)

Find Next Greater Element using Stack:

The idea is to store the elements for which we have to find the next greater element in a stack and while traversing the array, if we find a greater element, we will pair it with the elements from the stack till the top element of the stack is less than the current element.

illustration:
Below is the illustration of the above approach:



Follow the steps mentioned below to implement the idea:

- Push the first element to stack.
- Pick the rest of the elements one by one and follow the following steps in the loop.
    - Mark the current element as **next.**
    - If the stack is not empty, compare top most element of stack with **next**.
    - If **next** is greater than the top element, Pop element from the stack. **next** is the next greater element for the popped element.

    - Keep popping from the stack while the popped element is smaller than **next. next** becomes the next greater element for all such popped elements.
- Finally, push the **next** in the stack.
- After the loop in step 2 is over, pop all the elements from the stack and print **-1** as the next element for them.

Below is the implementation of the above approach:

C++Java

```
// A Stack based C++ program to find next
// greater element for all array elements.
#include <bits/stdc++.h>
using namespace std;

/* prints element and NGE pair for all
elements of arr[] of size n */
void printNGE(int arr[], int n)
{
    stack<int> s;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        /* if stack is not empty, then
           pop an element from stack.
           If the popped element is smaller
           than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */

        while (s.empty() == false && s.top() < arr[i]) {
            cout << s.top() << " --> " << arr[i] << endl;
            s.pop();
        }

        /* push next to stack so that we can find
        next greater for it */
        s.push(arr[i]);
    }
```

```
    /* After iterating over the loop, the remaining
    elements in stack do not have the next greater
    element, so print -1 for them */

    while (s.empty() == false) {
        cout << s.top() << " --> " << -1 << endl;
        s.pop();
    }
}

/* Driver code */
int main()
{
    int arr[] = { 11, 13, 21, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

Time Complexity: O(N)
Auxiliary Space: O(N)