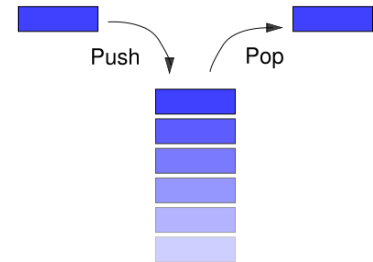


# STACK

A stack is a data structure that uses last-in, first-out (LIFO) ordering and allows reading and writing on the top element only.

## I. Language Agnostic Overview

The stack is perhaps the most fundamental of all data structures used in computer science. It has been adapted to store everything from simple numbers, to user-defined data types, to the entire state of running programs. The stack was developed over 50 years ago, making it one of the oldest ideas in the relatively young field of computer science.



The two included implementations of a stack, based either on a list or an array, were taken from the data structures textbook “Data Structures and Algorithm Analysis in Java” by Mark Allen Weiss.

Implementations of stacks are typically required to satisfy the following constraints:

1. You cannot pop elements from an empty stack.
2. Random element access is not allowed – only the “top” element can be read or removed, and new elements can only go on top.

There are a number of fundamental operations that can be performed on any stack:

- Empty() – This will destroy the contents of the stack, leaving it empty.
- Push(Element) – This puts a new element on top of the stack.
- Pop() – This removes the top element from the stack and returns it to the user.
- Top() – This returns the top element to the user, but does not affect the stack.
- IsEmpty() – This returns true if the stack is empty, otherwise it returns false.

## II. Java Specific Overview:

There are two ways to implement a stack, based on a list or based on an array.

### IIa. Array Based Implementation

The array that the stack is built on is initialized to either a default or a user-defined size depending on which constructor is called. New elements are pushed onto the stack by incrementing the variable `topOfStack`, and then inserting them into the newly incremented position. Elements are popped by setting the top object to null and then decrementing `topOfStack`.

There are two fields which are used in the invariants:

- `theArray` – This is the array of objects on top of which the Stack class is built. It stores the actual data which is accessed and modified by the push and pop methods of the Stack class.

- **topOfStack** – This is an integer which stores the count of the elements in the stack. Only the value which is stored at this key can be read.

So, the readable element is stored at `theArray[topOfStack]` and new elements are pushed at `theArray[++topOfStack]`.

### **push**

- This method first checks to make sure there is enough room on the stack for the new element. If there is, it inserts (pushes) a new element onto the top of the stack. If there is not enough room, an overflow exception is thrown.
- The preconditions verify that that the underlying array is not null, that its size is at least one, that the stack is not full, that the number of elements hasn't somehow been set to a negative value, and that the number of elements is not larger than the size of the underlying array. The postcondition verifies that the element was pushed successfully.

### **pushPop**

- This method pushes and pops an element. This has no effect on the permanent state of the stack, as the element is removed immediately after being added. It is used to test the functionality of the stack.
- The preconditions verify that that the underlying array is not null, that its size is at least one, that the stack is not full, that the number of elements hasn't somehow been set to a negative value, and that the number of elements is not larger than the size of the underlying array.

### **isFull**

- This method checks to see if the underlying array is full. If so, it returns true, if not, it returns false.
- This method has no invariants, rather it is used in the invariants of other methods.

## **IIb. List Based Implementation**

To construct a new stack with a list based implementation, the `topOfStack` pointer is initialized to null. New elements are pushed onto the stack by first making the new element point to whatever is currently at the top of the stack, and then assigning the `topOfStack` variable to point at the new element. Popping simply consists of assigning the `topOfStack` variable to the element that the current top points to – in effect making the second element become the first. It should again be noted that this implementation is generic to any object or primitive.

There is only one field checked by the invariant:

- topOfStack – This is an object of the ListNode data type. It points to the element on the top of the stack.

So, the top of the stack can be accessed by reading topOfStack.element.

### **Push**

- |                                                                                                                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>a. This method puts a new value on the top of the stack.</p> <p>b. The precondition verifies that the stack is not cyclic. The postconditions verify that the stack has not become cyclic, and that the element that was to be pushed is now on the top of the stack.</p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### **isAcyclic**

- |                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>a. This method traverses the stack and verifies that the same node isn't in the stack twice – that the stack doesn't point to itself at any point.</p> <p>b. This method has no invariants, rather it is used in the invariants of other methods.</p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### **Metrics:**

Lines of code for implementation: 114

Methods for implementation: 24

Lines of code for invariants: 45

Methods for invariants: 28

### **References:**

Wikipedia's entry on Stack data structures: [http://en.wikipedia.org/wiki/Stack\\_%28data\\_structure%29](http://en.wikipedia.org/wiki/Stack_%28data_structure%29)

Data Structures, Algorithms and Applications in Java by Sartaj Sahni. Chapter 9: “Stacks.” pgs 302-347.

- 9.3 “Array Representation.” pgs 307-315
- 9.4 “Linked Representation.” pgs 316-317

Algorithms in Java, 3<sup>rd</sup> Edition by Robert Sedgewick. Sections 4.4 – 4.5: “Abstract Data Types.” 148-157.

- 4.4 “Stack ADT Implementations.” pgs 148-153
- 4.5 “Generic Implementations.” pgs 154-157