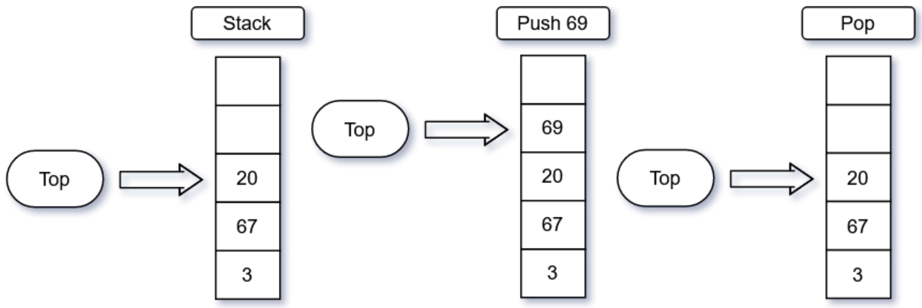
Introduction:

For those who do not know, the Stack data structure follows the **LIFO** (Last in First out) system. But what is that? Basically, what the **LIFO** system gives us is the order in which the elements can interact with the Stack. Following this system, we know that the first element that the user inserts on the Stack will be the last one getting out. The opposite is also true, the last element that the user inserts on the Stack is the first to leave.

Figure 1 - Stack example



In Figure 1 we can see how the two most important operations of the Stack work, *push* and *pop*. All these operations happen on the top part of the array. In the first example the Stack has two free positions and the top element is “20”. On the second one there is an insertion action (***push***) of the element “69”. It is important to notice that the top pointer moves up every time we insert a new element. On the last example there is a removal action (***pop***). This operation does not need input about the element one wants to remove, that is because the *pop* operation always removes the top element.

Examples:

There are many ways of explaining this basic data structure using real life analogies, we will give you two. Imagine you have five books, and you start stacking them up, the basic operations that one can do with those books are to put another book on the stack or simply take one out. Knowing that you can only take the top book of the stack out, the first book (the base) will, clearly, be the last one getting out. This simple example illustrates how the two basic operations of the Stack work.

Let us jump into the second example, when you are browsing through internet websites, even though you do not know it, there are two Stacks working to make your searching possible and efficient. The first Stack (***left stack***) will keep track of all the websites you have visited. So, every time you search a new site, the link of the website you were on goes to the ***left stack***. You may be wondering about the function of the second Stack (***right stack***) I have mentioned before. The second Stack exists to make it possible to move back and forth in the websites. Imagine you are on Facebook and you do a back click to your previous visited website (let us imagine it was Spotify), what happens with the Stacks? Facebook (the current website) would go to the ***right stack***, and Spotify, that as on the ***left stack***, would go to the current website position. Now if you do a forward click, the logic is the opposite. Spotify goes to the ***left stack*** and Facebook, that was on the ***right stack***, goes into the current website position.

Basic methods:

Push (insertion) – The insertion algorithm is quite simple. First check if the Stack is full, in case it is, produce an error and stop. If the Stack has free positions, the other only requirement is to insert the element at the end of the collection data type you are using (usually arrays).

Pop (removal) – The removal algorithm is the following. First check if the Stack is empty, if it is, raise an error. Otherwise all you need to do is take the last element of the collection data type out and return it.

Peek (Consulting) – The peek function enables the user to see the last element inserted on the Stack, without removing it.

isEmpty – Returns “True” if the Stack is empty and “False” if not.

isFull – Returns “True” if the Stack is full and “False” if not.

Clear – Restarts the Stack.

Size – Returns the number of elements in the Stack.

All the methods of the Stack are very efficient, because their time complexity is O(1). This makes the Stack an exceptionally good option to solve some basic problems fast. In the next section we are going to present two different paths you can follow to implement your Stack.

Implementation:

In this part we are going to approach two different ways of implementing the Stack. The first method requires a list data type (***StackList***). The list is going to be initialized in the class constructor method. All the other methods will interact with the list, making it possible to add and remove elements from it.

The second method, instead of using a list, uses a numpy array (***StackArray***). This structure difference is going to make some changes in the way we implement the *Stack*. When using the array implementation, it is necessary to use a pointier to the position of the last inserted element. This happens because, when we want to add and element to a python list (***append***), the element is added to the end by predefinition, while in the array you need to know the next empty position to insert at the end.

This second method has some advantages, such as, being more efficient. When an array is created there is a reserved space in memory for it, but the list does not work this way. The list is like an array, but it only keeps references to other spaces in the memory, making it less efficient. On the other hand, the array is limited, because in order to create it, the user must input the desirable size. Using the list implementation is a smart choice when the elements to insert are from different natures (*string, integer, float, tuple…*), since the array can only store elements of the same type. Although both implementations are basic, the simplest is the list implementation, because it does not need a pointer to be able to serve the Stack's purpose.