Stacks

Step 1

The next **Abstract Data Type** we will discuss is the **Stack**. If you have ever "stacked" up the dishes after dinner to prepare for washing, you have (potentially unknowingly) used a **Stack**. When you are washing the dishes after dinner, you add unwashed dishes to the top of the stack of dishes, and when you want to wash the next dish, you take one off from the top of the stack of dishes. With a **Stack**, the *first* element to come out is the *last* one to have gone in. Because of this, the **Stack** is considered a **"Last In, First Out" (LIFO)** data type.

Formally, a **Stack** is defined by the following functions:

- **`push(element)`:** Add `element` to the top of the Stack
- **`top()`:** Look at the element at the top of the Stack
- **`pop()`:** Remove the element at the top of the Stack

STOP and Think: Do these functions remind us of an **Abstract Data Type** we've learned about?

Step 2

Just like with the **Queue**, we can use a **Deque** to implement a **Stack**: if we implement a **Stack** with a **Deque** as our backing structure (where the **Deque** would have its own backing structure of either a **Doubly Linked List** or a **Circular Array**, because as you should recall, a **Deque** is an **ADT**), we can again simply re-use the functions of a **Deque** to implement our **Stack**. For example, say we had the following **Stack** class in C++:

```
class Stack {
    private:
        Deque deque;
    public:
        bool push(Data element);
        Data top();
        void pop();
        int size();
};
```

We could very trivially implement the **Stack** functions as follows:

```
bool Stack::push(Data element) {
    return deque.addBack(element);
}
```

```
Data Stack::top() {
    return deque.peekBack();
}
```

```
void Stack::pop() {
    deque.removeBack();
}
```

```
int Queue::size() {
    return deque.size();
}
```

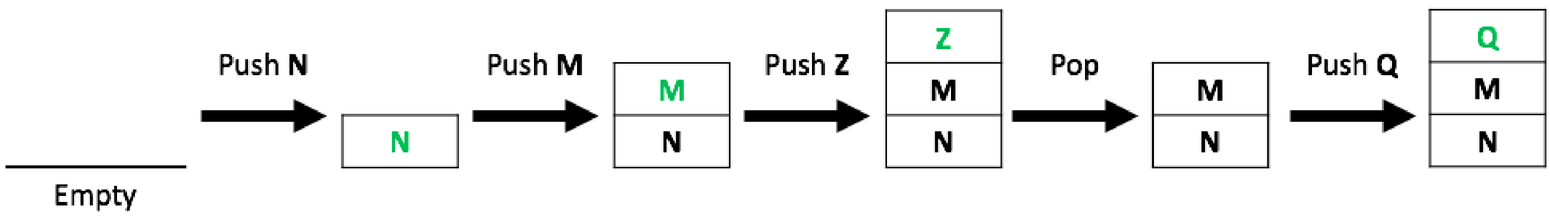
Of course, as we mentioned, the **Deque** itself would have some backing data structure as well, but if we use our **Deque** implementation to back our **Stack**, the **Stack** becomes extremely easy to implement.

Watch Out! Notice that, in our implementation of a **Stack**, the `pop()` function has a `void` return type, meaning it *removes* the element on the top of the **Stack**, but it does not *return* its value to us. This is purely an implementation-level detail, and in some languages (e.g. Java), the `pop()` function removes *and* returns the top element, but in other languages (e.g. C++), the `pop()` function *only removes* the top element without returning it, just like in our implementation.

STOP and Think: In our implementation of a **Stack**, we chose to use the `addBack()`, `peekBack()`, and `removeBack()` functions of the backing **Deque**. Could we have chosen `addFront()`, `peekFront()`, and `removeFront()` instead? Why or why not?

Step 3

Below is an example in which we push and pop elements in a **Stack**. Note that we make no assumptions about the implementation specifics of the **Stack** (i.e., we don't use a **Linked List** nor a **Circular Array** to represent the **Stack**) because the **Stack** is an **ADT**.



Step 4

EXERCISE BREAK: What is the worst-case time complexity of **adding *n* elements** into a **Stack**, given that we are using a **Deque** as our backing structure in our implementation and given that we are using a **Doubly Linked List** as the backing structure of our **Deque**?

To solve this problem please visit https://stepik.org/lesson/28873/step/4

Step 5

EXERCISE BREAK: Say I have a stack of integers *s*, and I run the following code:

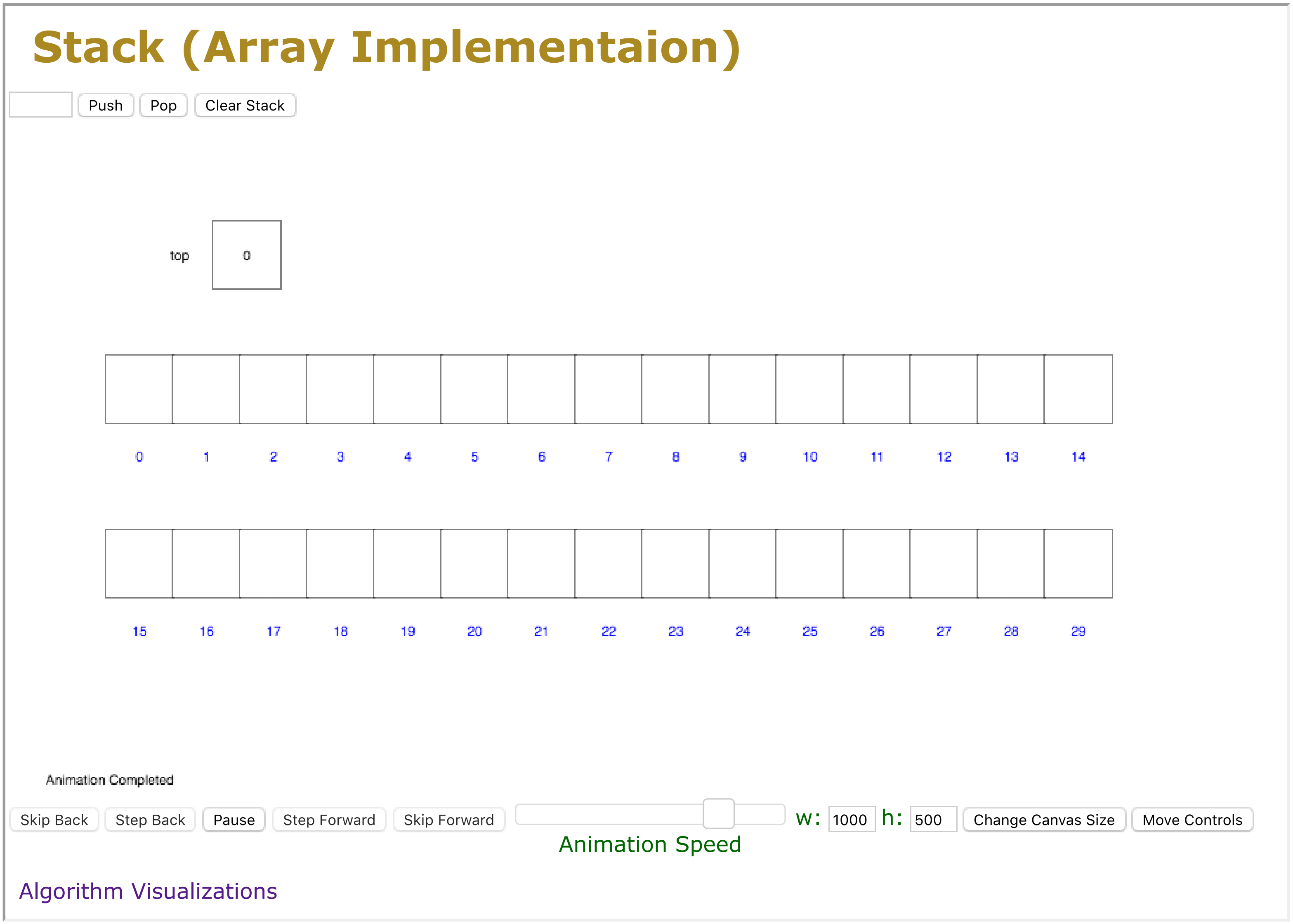
```
for(int i = 0; i < 10; i++) {
    s.push(i);
}
s.pop();
s.push(100);
s.push(200);
s.pop();
s.pop();
s.pop();
```

What is the number at the top of the stack after running this code? In other words, what would be returned if I call **`s.top()`**?

To solve this problem please visit https://stepik.org/lesson/28873/step/5

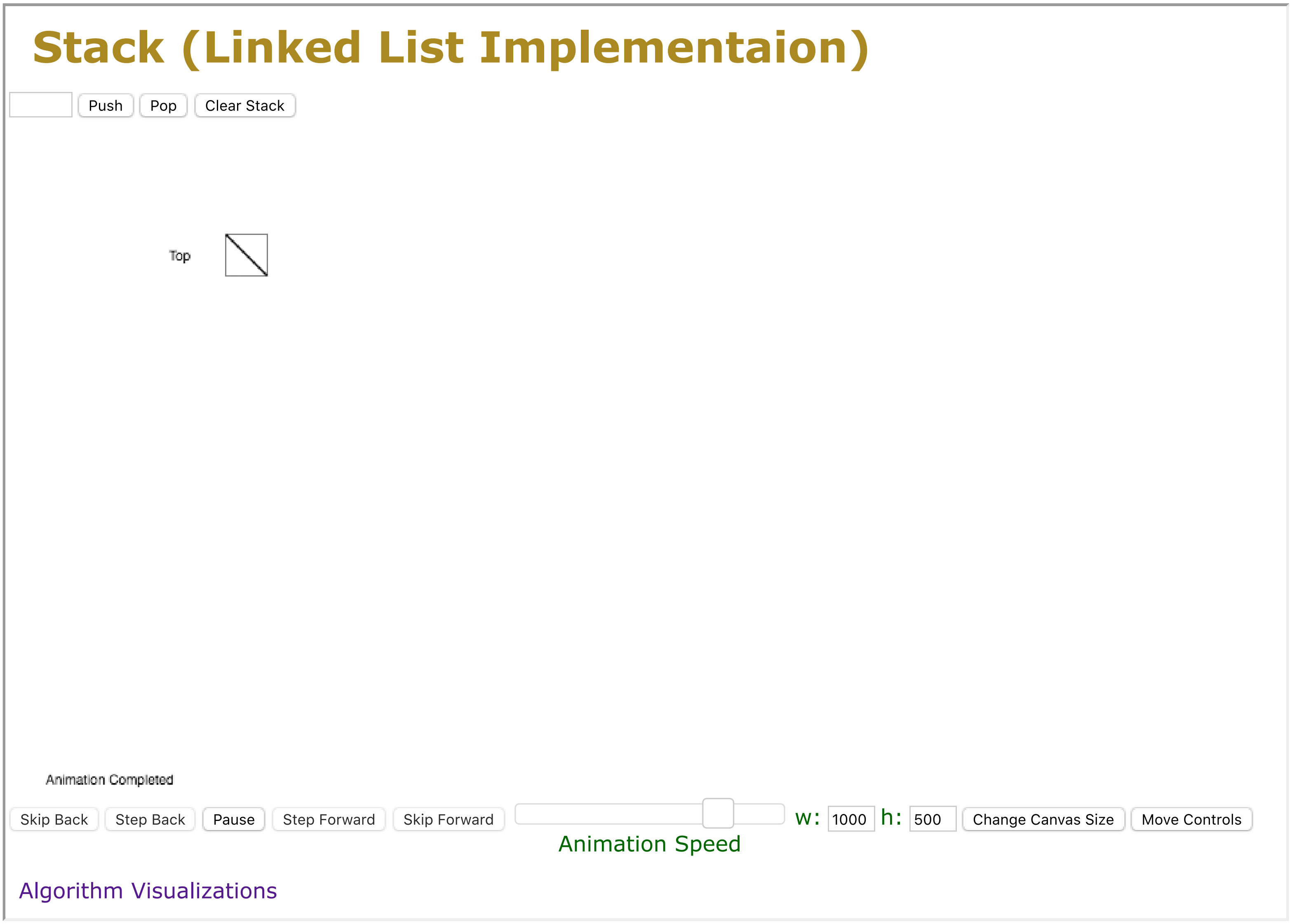
Step 6

Below is a visualization of a **Stack** backed by a **Circular Array**, created by David Galles at the University of San Francisco.



Step 7

Below is a visualization of a **Stack** backed by a **Linked List**, created by David Galles at the University of San Francisco.



Step 8

Despite its simplicity, the **Stack** is a very powerful **ADT** because of the numerous applications to which it can be applied, such as:

- Storing student exams that need to be graded
- Keeping track of the evaluation of smaller expressions within a large complex mathematical expression
- "Graph" exploration via the "Depth-First Search" algorithm (which will be covered in the "Graphs" section of this text)

This concludes our discussions about the introductory **Data Structures** and **Abstract Data Types** we will be covering in this text. We hope that the knowledge you acquired will help you implement them as well as to give you some intuition to help you understand more advanced data structures that we will encounter in the later sections of the text.