Stacks

A stack if an ordered collection of elements where elements are only added and removed from the same end. It is LIFO. An analogy is a stack of plates in a kitchen - you add plates or remove plates from the top of the pile. A software example is using a stack to keep track of the history of your current browser's tab. Let's say you're on site `A`, and you click on a link to go to site `B`, then from `B` you click on another link to go to site `C`. Every time you click a link, you are adding to the stack - your history is not `[A, B, C]`. When you click the back arrow, you are "removing" from the stack - click it once and you have `[A, B]`, click again and you have `[A]`.

Stacks are very simple to implement. Some languages like Java have built-in stacks. In Python, you can just use a list `stack = []` and use `stack.append(element)` and `stack.pop()`. In fact, any dynamic array can implement a stack. Inserting is called **pushing** and deleting is called **popping**. **peek** returns the element at the top of the stack without removing it.

If you use a dynamic array (the most common and easiest way) to represent a stack, then the time complexity of your operations is the same as that of a dynamic array. $O(1)$ push, pop, and random access, and $O(n)$ search. Sometimes, a stack may be implemented with a linked list with a tail pointer.

Recursion is actually done using a stack.

For problems, a stack is a good option wherever you recognize the LIFO pattern. Usually, there will be some component of the problem that involves elements in the input interacting with each other. Interacting could mean matching elements together, querying some property such as "how far is the next largest element", evaluating a mathematical equation given as a string, just comparing elements against each other, or some other abstract interaction.

Interface guide

Python3

```
# Declaration: we will just use a list
stack = []

# Pushing elements:
stack.append(1)
stack.append(2)
stack.append(3)
```

```
# Popping elements:
stack.pop() # 3
stack.pop() # 2

# Check if empty
not stack # False

# Check element at top
stack[-1] # 1

# Get size
len(stack) # 1
```

JavaScript

```
// Declaration: we will just use a list
let stack = [];

// Pushing elements:
stack.push(1);
stack.push(2);
stack.push(3);

// Popping elements:
stack.pop(); // 3
stack.pop(); // 2

// Check if empty:
!stack.length; // false

// Check element at top
stack[stack.length - 1]; // 1

// Get size
stack.length; // 1
```

Java

```
// Declaration: Java supports multiple implementations, but we will be using
// the Stack interface with the Stack implementation. Specify the data type
Stack<Integer> stack = new Stack<>();

// Pushing elements:
stack.push(1);
stack.push(2);
stack.push(3);

// Popping elements:
stack.pop(); // 3
stack.pop(); // 2

// Check if empty
stack.empty(); // false

// Check element at top
stack.peek(); // 1

// Get size
stack.size(); // 1
```

C++

```
// Declaration: C++ supports multiple implementations, but we will be using
// std::stack. Specify the data type
stack<int> stack;

// Pushing elements:
stack.push(1);
stack.push(2);
stack.push(3);

// Popping elements:
// Note, unlike other languages, popping here does not return the popped
value
```

```
stack.pop();
stack.pop();

// Check if empty
stack.empty(); // false

// Check element at top
stack.top(); // 1

// Get size
stack.size(); // 1
```