# Stacks and Queues

## Stacks

1. https://www.coursera.org/learn/data-structures/lecture/UdKzQ/stacks

**Stack**:

* ***LIFO*** (*Last In First Out*) data structure (also known as *LIFO* queues) (abstract data type).
* Can be implemented with either an **array** or a **linked list**.
* Each stack operation is **O(1)**: Push,Pop, Top, Empty.
* Example: a stack of books, we can take the top element, and put another element at the top, but we can’t take the element at the bottom.

It has the following operations.

* **Push(Key)**: adds key to collection.
* **Key Top()**: returns mostrecently-added key.
* **Key Pop()**: removes and returns most recently-added key.
* **Boolean Empty()**: are there any elements?

### Example

A stack allows to check if an expression has balanced brackets:

* Input: A string str consisting of ‘(‘, ‘)’, ‘[‘,‘]’ characters.
* Output: Return whether or not the string’s parentheses and square brackets are balanced.

Example:

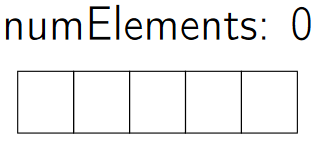
* Balanced:“([])[]()”,“((([([])]))())”
* Unbalanced:“([]]()”“][”

#### IsBalanced(str)

|  |  |
| --- | --- |
| Stack *stack* | *Create a stack.* |
| for *char* in *str*: | *Go through every character in the string.* |
| if *char* in [‘(‘, ‘[‘]:  *stack*.Push(*char*) | *If we have an opening character,*  *push the character to the stack.* |
| else: | *If it isn’t an opening character,* |
| if *stack*.Empty():  return False | *Check if the stack is empty, if it is then it isn’t balanced and we return False.* |
| *top*←*stack*.Pop() | *Pop the top element off,* |
| if (*top* = ‘[‘ and *char* != ‘]’) or (*top* = ‘(‘ and *char* != ‘)’):  return False | *Check if the top element matches the character that we got (so if the top is a ‘[’ and we read a ‘]’ then it matches) and we continue.* |
| return *stack*.Empty() | *Once we’ve run through all of the characters, we check again if the stack is empty.* |

### Stack Implementation with Array.

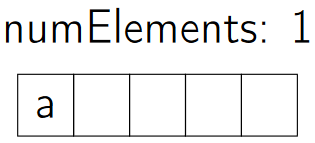
Created by allocating an array of some maximum stack size (example: 5).



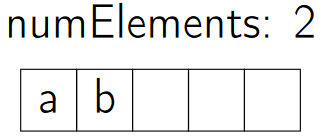
* **Advantage of using an array**:
  + All operations are done in constant time *O(1)*.
* **Disadvantages of using an array**:
  + We have a maximum size based on the array we initially allocated.
  + Potential wasted space (unused indices).

#### Push O(1)

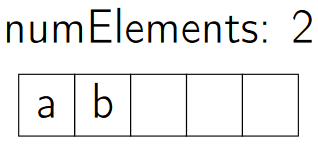
1. When we *push* an element (example: a), we put it at the end of the array we got so far.



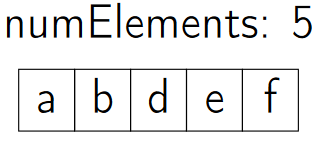
* Pushing (or *appending*) another element (b) would put it at the end of the array.



1. The stack would like this:

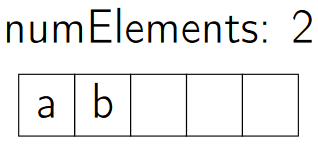


**Note**: pushing more elements than the original array allows would throw an error.



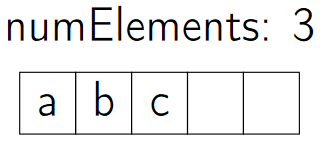
#### Top O(1)

1. Returns the top element of the array (would return b) but it *doesn’t remove the element*.



#### Pop O(1)

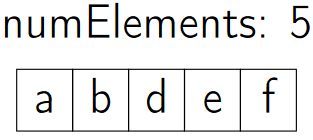
1. Returns and removes the last element of the stack (example: c).

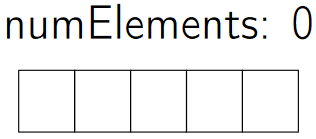


**Note**: popping when the array is empty would return an error.

#### Empty O(1)

1. Check whether the number of elements is greater than zero.



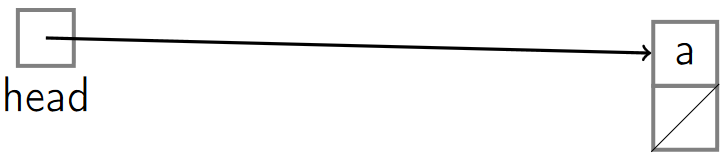


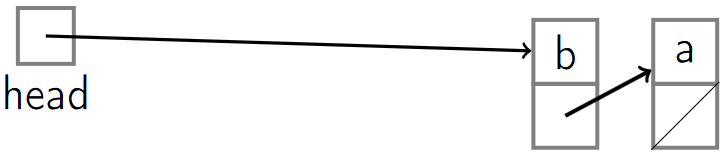
### Stack Implementation with Linked-List.

* **Advantage of using a linked list**:
  + All operations are done in constant time *O(1)*.
  + There’s no limit to the amount of elements that can be added (as long as there’s available memory).
  + No wasted space because only the used space is allocated.
* **Disadvantages of using a linked list**:
  + Using array, each element size is just big enough to store the key, using a linked-list we also have to store a pointer.

#### Push O(1)

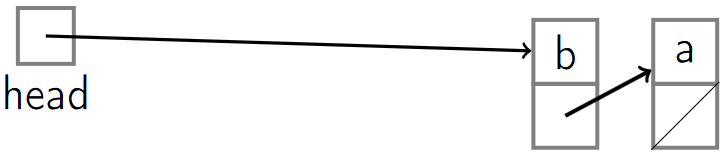
1. Pushing an elements pushes it to the front (*PushFront()*).
2. The head is updated to point to the newly pushed element.





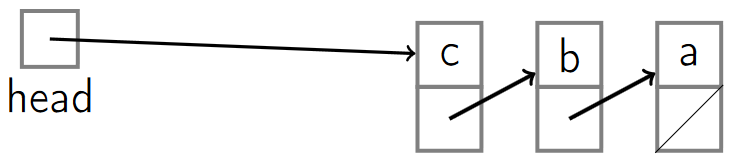
#### Top O(1)

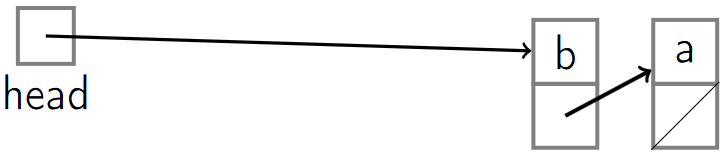
1. Returns the top element of the array (would return b) but it *doesn’t remove the element*. (*TopFront()*).



#### Pop O(1)

1. Returns and removes the last element of the stack (example: c). (*TopFront(), PopFront()*).





#### Empty O(1)

1. Check if the head points to *null*.

