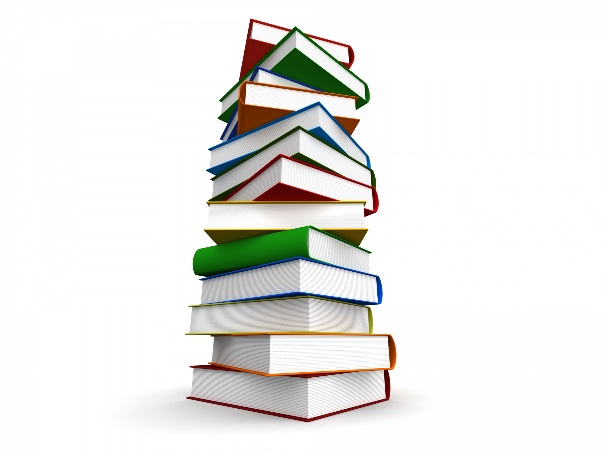
**Stack**

Stack is a Sequential access data structure. Because if we want to access an item in Stack, we should follow an order/sequence. Accessing an item depends on other items.

In stack we add elements and remove elements based on LIFO (Last in first out) order.

So, stack is a sequential data structure in which we add elements and remove elements according to the LIFO order.

The real-life example of stack is books in library. These are stacked on top of each other. If we want to remove a book which in the middle of the stack first, we should remove all the books which are added after that book.





LI (Last in)

FO (First out)

In stack there is only one way to **IN** and **OUT**

The most common and widely used Stack methods are.

1. Push
2. Pop
3. Peek
4. Contains
5. isFull
6. isEmpty

**Push**

Push is used to add the element/object into the stack. When we push an object into the stack its size will increase by 1 dynamically. Push will not return anything.

Now create a stack and push some objects into the stack.

demo\_stack. push(‘hi’)

demo\_stack. push(‘hello’)

demo\_stack. push(‘world’)

fddsfdsfsd

Size=3

world

Size=2

hello

hi

Size=1

demo\_stack

**Pop**

Pop is used to remove the object/element from the stack. Pop will remove the item which is at the top of the stack. Pop returns the removed element.

world

demo\_stack. pop ()

demo\_stack. pop ()

demo\_stack. pop ()

fddsfdsfsd

hello

hi

Size=0

**Peek**

Peek is used to get the top element in the stack without removing it.

fddsfdsfsd

Size=3

world

Size=2

hello

hi

Size=1

demo\_stack

demo\_stack. Peek () ====🡺 o/p: world

when we want to know the element at the top of the stack without removing it, we will use Peek ().

**Contains**

This method is used to search through the stack. when we want to know whether a particular element is present in the stack or not, then we will use contains () method.

If the element is there in the stack it will return True, otherwise returns False.

fddsfdsfsd

Size=3

world

Size=2

hello

hi

Size=1

demo\_stack

demo\_stack. Contains(“hi”) ===🡺 True.

**isFull**

isFull is used to check whether a stack is Full or not. Returns True if it is full otherwise, returns False.

fddsfdsfsd

Size=3

world

Size=2

hello

hi

Size=1

demo\_stack

demo\_stack. IsFull () ====🡺 False

**isEmpty**

isFull is used to check whether a stack is Empty or not. Returns True if it is full otherwise, returns False.

fddsfdsfsd

Size=3

world

Size=2

hello

hi

Size=1

demo\_stack

demo\_stack. IsEmpty () =====🡺 False

**Time complexity of Stack**

**1. Accessing**

The time complexity for accessing an element in a stack is O(n). because if we want to access an element in a stack first, we need to pop all the elements on top of that element.

We have 5 elements in stack, in worst case scenario we want to access 1st element in the stack, then we need to remove all the 4 elements on top of the 1st element then access the 1st element. The number of operations required are 5. Big-O = O (5).

So, the time complexity is O(n).

**2. Searching**

Similar like accessing, the time complexity for searching through the stack is also O(n).

**3. Inserting**

The time complexity for inserting an element into the stack is O (1). Because when we push a new element into the stack its always going to be add at top the stack.

**4. Deleting**

Similar like inserting, the time complexity for deleting an element from the stack is O (1).

Because when we pop the element it is always the top element.

**Report of Stack**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accessing | Searching | Inserting | Deleting |
| Time complexity | O (n) | O (n) | O (1) | O (1) |

**Note:**

Stack is the back bone for recursion functions.

Stacks are also used in Undo/Redo in applications like paint, word and Back-paging in web browsers to navigate to previous web page.