Stack

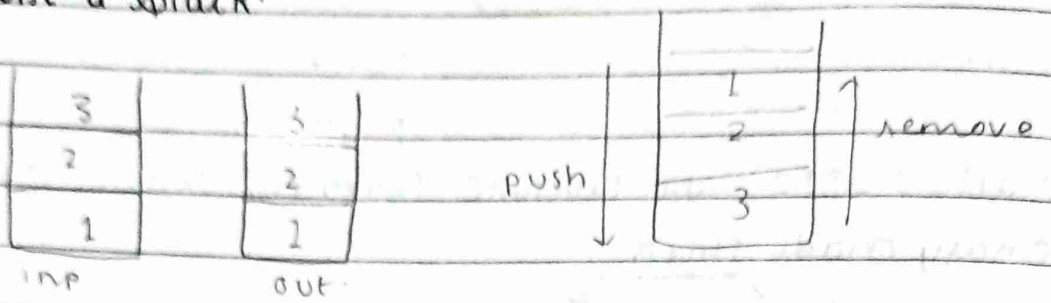
* Stack is LIFO Data Structure (Last In First Out)
* memory made stack
  - implicit stack
  - due to recursion or backtracking.
* self made stack → explicit stack

Stacks Using Arraylist:-

```
public static boolean isEmpty(){
        return list.size==0;
        }

public static void push(){
        list.add(data);
    }

public static int pop(){
   int top = list.get(list.size()-1);
   list.remove(list.size()-1);
   return top;
   }

public static int peek(){
      return list.get(list.size()-1);
      }
```

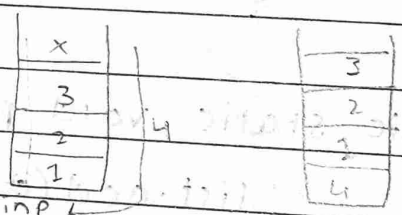
* when we can't use extra memory for implementation then recursion is one of the ways to use.

Q. Reverse a stack.

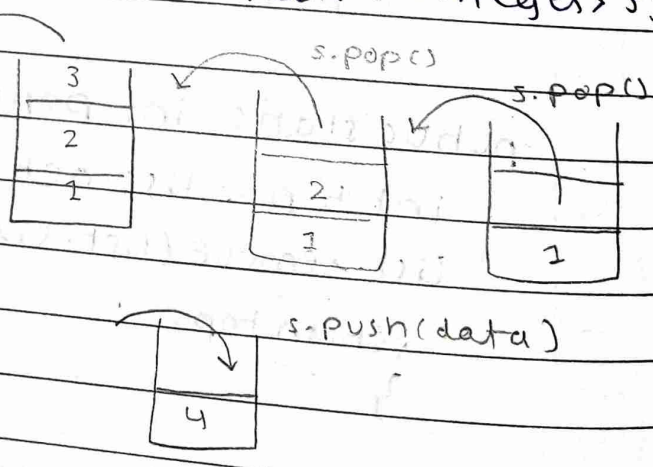



```
public static void PushAtBottom (Stack <Integer> s,
  int data) {
   if(isEmpty()){
        s.push(data);
        return;
    }
   int top = s.pop();
   PushAtBottom (s,data);
   s.push(top);
  }
```

first we will empty stack one by one and when the stack gets empty we will push the given element and then the previous elements in same order.
we will do this with the help of recursion.

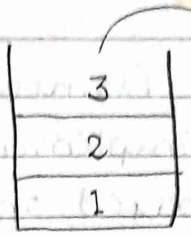



```
public static void reverseStack ( Stack < Integer> s){
  if(isEmpty()) {
      return;
    }
  int top = s.pop();
  reverseStack (s);
  PushAtBottom (s, top);
  }
```



Operations on stack:-

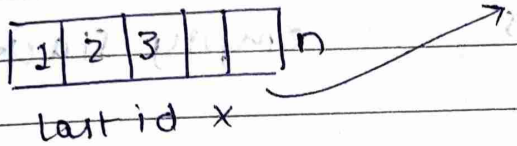
Push → O(1)
Pop → O(1)
Peek → O(1)
↳ Top in C++

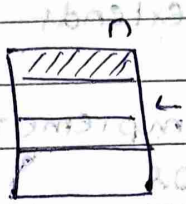


Last In First Out.

3 Approaches to implement stack from scratch:-

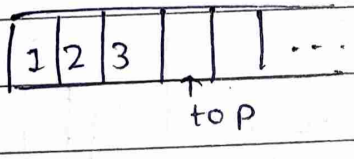
1. **Array** → Avoid Arrays ✗
   - fixed size :(





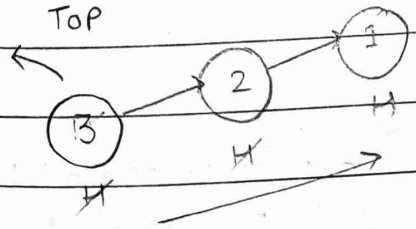
stackfull (cuss == n)

2. **Arraylist** → variable size. (dynamic)



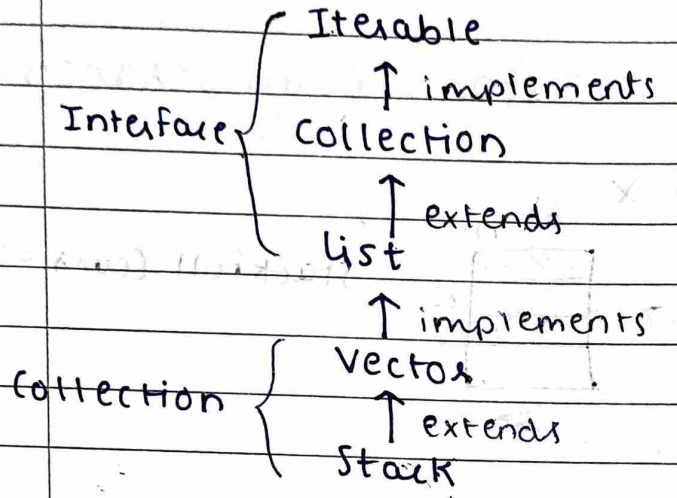
The empty space is always the top.

3. **Linked list** → variable size.



Always make the last node as head (Top) cause we have to operate the operations as O(1)

Java collection framework provides a stack class that models and implements a stack Data structure.
The class is based on the principle of last in First out.



The class supposts one default constructor Stack() which is used to create an empty stack.