# Assignment 2 - Part 1

Q - 1) Assume that we have one single array A of size N, where N is an even value, and that the array is not expandable. We need to implement 2 stacks. Develop a well-documented pseudo-code that shows how these two stacks can be implemented using this single array. There are two cases you must consider: Case I: Fairness in space allocation to the two stacks is required. In that sense, if Stack1 for instance use all its allocated space, while Stack2 still has some space; insertion into Stack1 cannot be made, even though there are still some empty elements in the array; Case II: Space is critical; so you should use all available elements in the array if needed. In other words, the two stacks may not finally get the same exact amount of allocation, as one of them may consume more elements (if many push() operations are performed for instance into that stack first). For each of the two cases:

**A. Briefly describe your algorithm;**

**Answer:**

- My implementation of the above-mentioned abstract data type i.e. (2 stacks within an array) stems from the fact that there are few catches to be taken care of while implementing a stack. The things to be taken care of are that here there would be two top pointers, two pop methods, two push methods, and two of isEmpty and isFull methods. The ideal logic here would be to push from the back and forth of the array while you reach the other end or collide with some element inserted(pushed) by the other stack in the array. As soon as this happens, we get a stack overflow exception.
- The other thing to keep in mind is that my implementation for the two stacks within a single array is dynamic in nature. For instance, let's assume the size of the array is 100 and we reach a point where the array becomes full. At this point, if either one of the stacks removes(pop) an element, the recently empty index would be available for the stacks, whoever inserts an element first would take the space and thus the space allocation for each stack would change dynamically.
- **Just an important thing to keep in mind here is that there are different methods to push and pop the elements depending on the stack we are inserting or deleting.**
- Thus, this ADT has been designed keeping in mind the critical nature of the space and thus is efficient. Each operation is O(1) Time, which makes it very efficient for usage.

**B. Write, in pseudocode, the implementation of push(), pop(), size(), isEmpty() and isFull();**

**Answer:**

**Method pushFront(element)**
Input: An element value of any type, here the type is Integer.
Output: Null, with a value of the element being pushed into stack 1.

```
pushFront(element)
        If (topFront < stack1.size() - 1 and stack[topFront] = -Infinity)
                stack[topFront] ← element
                topFront ← topFront + 1
```

```
            Return
        Raise new Exception("Stack 1 Overflowed")
```

**Method pushRear(element)**
Input: An element value of any type, here the type is Integer.
Output: Null, with a value of the element being pushed into stack 2.

```
pushRear(element)
        If (topFront > 1 and stack[topRear] = -Infinity)
                stack[topRear] ← element
                topRear ← topRear - 1
                Return
        Raise new Exception("Stack 2 Overflowed")
```

**Method popFront()**
Input: Null
Output: Returns the value of the element from the stack 1 if not empty else raises an exception

```
popFront()
        If (topFront > 0)
                topFront ← topFront - 1
                elementToBePopped = stack1[topFront]
                stack1[topFront] ← -Infinity
                Return elementToBePopped
        Raise new Exception("Stack 1 Underflowed")
```

**Method popRear()**
Input: Null
Output: Returns the value of the element from the stack 2 if not empty else raises an exception

```
popRear()
        If (topRear < stack.size() - 1)
                topRear ← topReart + 1
                elementToBePopped = stack2[topRear]
                stack1[topRear] ← -Infinity
                Return elementToBePopped
        Raise new Exception("Stack 2 Underflowed")
```

**Method size(stackType)**
Input: String indicating the "Front" or the "Rear" stack.

Output: An Integer value representing the size of the stack determined.

```
size(stackType)
If (stackType = “Front”)
        Return stack[topFront]
Else
        Return stack[topRear]
```

**Method isEmpty(stackType)**
Input: String indicating the “Front” or the “Rear” stack.
Output: A boolean value representing if the mentioned stack is empty or not.

```
isEmpty(stackType)
        If(stackType = “Front”)
                Return (topFront = 0)
        Else
                Return (topRear = nums.size() -1)
```

**Method isFull(stackType)**
Input: String indicating the “Front” or the “Rear” stack.
Output: A boolean value representing if the mentioned stack is full or not.

```
isFull(stackType)
        If(stackType = “Front”)
                If(topFront < nums.size() - 1 and stack1[topFront] is not equal to -Infinity)
                        Return False
                Else
                        Return True
        Else
                If(topRear > 1 and stack2[topRear] is not equal to -Infinity)
                        Return False
                Else
                        Return True
```

**C) What is the Big-O complexity for the methods in your solution? Explain clearly how you obtained such complexity.**

**Answer:**

- For all the methods mentioned above, the time complexities in terms of Big-O are gonna be O(1) as we do not use any extra space rather than storing the integer values in the stack data type.
- Also, in terms of extra pointers to keep track of the starting and ending positions of each stack, we just use two variables.
- Finally, no looping structures are involved in any of the methods, which makes the Big-O as well as Big-Ω complexities as O(1).

**D) What is the Big-Ω complexity of your solution? Explain clearly how you obtained such complexity.**

**Answer:**

- As the Big-O complexities for all the methods are simply O(1) as we do a couple of O(1) operations in all the methods, it is very clear that the Big-Ω complexities for all the methods would be definitely O(1).

**Q - 2)** Assume that we have one single array A of size N, where N is an even value, and that the array is not expandable. We need to implement 2 stacks. Develop a well-documented pseudo-code that shows how these two stacks can be implemented using this single array. There are two cases you must consider: Case I: Fairness in space allocation to the two stacks is required. In that sense, if Stack1 for instance use all its allocated space, while Stack2 still has some space; insertion into Stack1 cannot be made, even though there are still some empty elements in the array; Case II: Space is critical; so you should use all available elements in the array if needed. In other words, the two stacks may not finally get the same exact amount of allocation, as one of them may consume more elements (if many push() operations are performed for instance into that stack first). For each of the two cases:

**A) Write the pseudocode of the max() method.**

**Answer: Here, in order to understand the pseudocode for the max() method, I will also write the pseudocode for the push() method.**

**Method push(element)**
Input: An element to the inserted in the stack of the type integer here.
Output: Null, with an element being inserted into the stack.

```
push(element)
    newMax ← Dictionary[String - Integer]
    newMax.put(“max”, element)
    If(stack.size() > 0)
            lastMax  ← Dictionary[String - Integer](stack.get(stack.size() - 1))
            newMax.replace(“max”, Max(lastMax.get(“max”), element))
    stack.add(newMax)
    stack.add(element)
```

**Method getMax()**
Input: Null
Output: Returns the current maximum element in the stack if not empty.

```
getMax()
    Return Stack.get(stack.size() - 1).get("max")
```

**B) What is the Big-O complexity of your solution? Explain clearly how you obtained such complexity.**

**Answer:**

- The Big-O complexity of my implementation would be straight-forward. The time complexity would be equal to O(1) for all the methods as we do nothing but just do a few couple **O(1)** operations.
- However, the space complexity will be equal to **O(N)** as we keep an ArrayList and a Hashmap to keep track of the maximum elements in the stack currently.
- Therefore, Time: O(1) for all the methods implemented in the static class.
  Space: O(N), where N is the total number of elements being pushed into the stack.

**C) Is it possible to have all three methods (push(), pop(), and max()) be designed to have a complexity of O(1)? If no; explain why this is impossible. If yes; provide the pseudocode of the algorithm that would allow O(1) for all three methods (this time, you do not only need to think about it, but to actually give the pseudocode if you believe a solution is feasible!)**

**Answer:**

Yes, it is definitely possible to implement these methods in O(1) time complexity. Following the pseudocode for all the methods.

**Method push(element)**
Input: An element to the inserted in the stack of the type integer here.
Output: Null, with an element being inserted into the stack.

```
push(element)
    newMax ← Dictionary[String - Integer]
    newMax.put("max", element)
    If(stack.size() > 0)
        lastMax ← Dictionary[String - Integer](stack.get(stack.size() - 1))
        newMax.replace("max", Max(lastMax.get("max"), element))
```

```
    stack.add(newMax)
    stack.add(element)
```

**Method getMax()**
Input: Null
Output: Returns the current maximum element in the stack if not empty.

```
getMax()
    Return Stack.get(stack.size() - 1).get("max")
```

**Method pop()**
Input: Null
Output: Returns the last (top-most) element from the stack.

```
pop()
    maxStack.remove(maxStack.size() -1)
    Return stack.remove(stack.size() - 1)
```

**Q - 3) Is it possible to have a single tree with the following traversals? If yes, draw the tree. If not; explain which elements prevent you most from having one such tree.**

- **Each internal node of T stores a single character;**
- **A preorder traversal of T yields: E K D M J G I A C F H B L;**
- **A postorder traversal of T yields: D J I G A M K F L B H C E.**

**Answer:**

- **NO**, it is not possible to create a single tree having the above traversals.
- The most problematic nodes are the J-I-G-A as they would not meet the Pre-order requirements if constructed by Post-order traversal and vice versa.