1. An **extendable array** is a data structure that stores a sequence of items and supports the following operations.

   - **AddToFront**\(x\) adds \(x\) to the *beginning* of the sequence.
   - **AddToBack**\(x\) adds \(x\) to the *end* of the sequence.
   - **Lookup**\(k\) returns the \(k\)th item in the sequence, or **Null** if the current length of the sequence is less than \(k\).

Describe a **simple** data structure that implements an extendable array. Your **AddToFront** and **AddToBack** algorithms should take \(O(1)\) amortized time, and your **Lookup** algorithm should take \(O(1)\) worst-case time. The data structure should use \(O(n)\) space, where \(n\) is the **current** length of the sequence.

2. An **ordered stack** is a data structure that stores a sequence of items and supports the following operations.

   - **OrderedPush**\(x\) removes all items smaller than \(x\) from the beginning of the sequence and then adds \(x\) to the beginning of the sequence.
   - **Pop** deletes and returns the first item in the sequence (or **Null** if the sequence is empty).

   Suppose we implement an ordered stack with a simple linked list, using the obvious **OrderedPush** and **Pop** algorithms. Prove that if we start with an empty data structure, the amortized cost of each **OrderedPush** or **Pop** operation is \(O(1)\).

3. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array \(A[1..n]\) that stores the height of \(n\) buildings on a city block, indexed from west to east. Building \(i\) has a good view of Lake Michigan if and only if every building to the east of \(i\) is shorter than \(i\).

   Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

   ```
   GoodView(A[1..n]):
   initialize a stack S
   for i ← 1 to n
     while (S not empty and A[i] > A[Top(S)])
       Pop(S)
       Push(S, i)
   return S
   ```