COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Introduction

- How do Data Structures play a part in making computational tasks efficient?

Problem

Multiplying two n-bit numbers: Given two n-bit numbers, A and B, Design an algorithm to output $A \cdot B$.

Algorithm

`Karatsuba`(A, B)

- If $(|A| = |B| = 1)$ return$(A \cdot B)$
- Split A into A_L and A_R
- Split B into B_L and B_R
- $P \leftarrow$ `Karatsuba`(A_L, B_L)
- $Q \leftarrow$ `Karatsuba`(A_R, B_R)
- $R \leftarrow$ `Karatsuba`$(A_L + A_R, B_L + B_R)$
- return(`Combine`(P, Q, R))

- Recurrence relation: $T(n) \leq 3 \cdot T(n/2) + cn$; $T(1) \leq c$.
- What is the solution of this recurrence relation?
  $T(n) \leq O(n^{\log_2 3})$

Problem

Given an array of unsorted integers, output a sorted array.

Algorithm

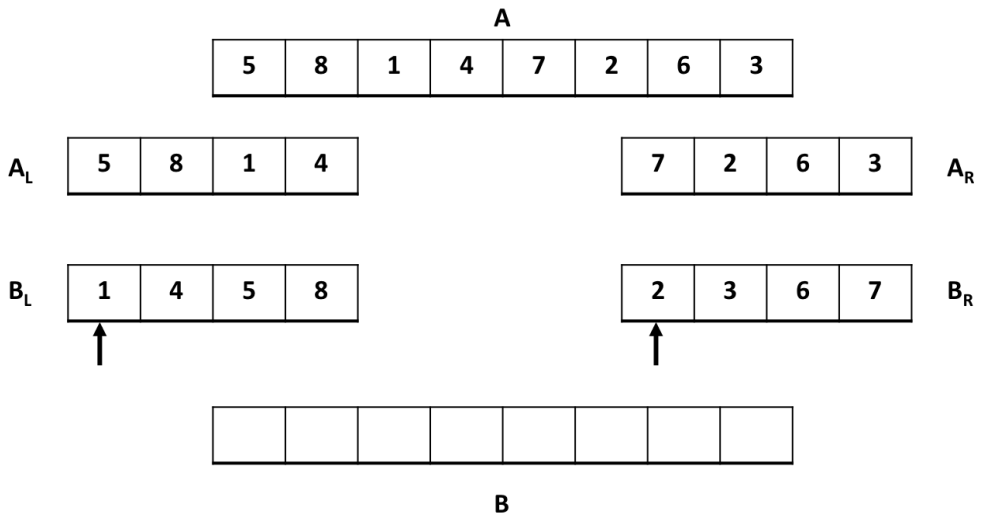
`MergeSort`(A)

- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

Algorithm

`MergeSort`(A)

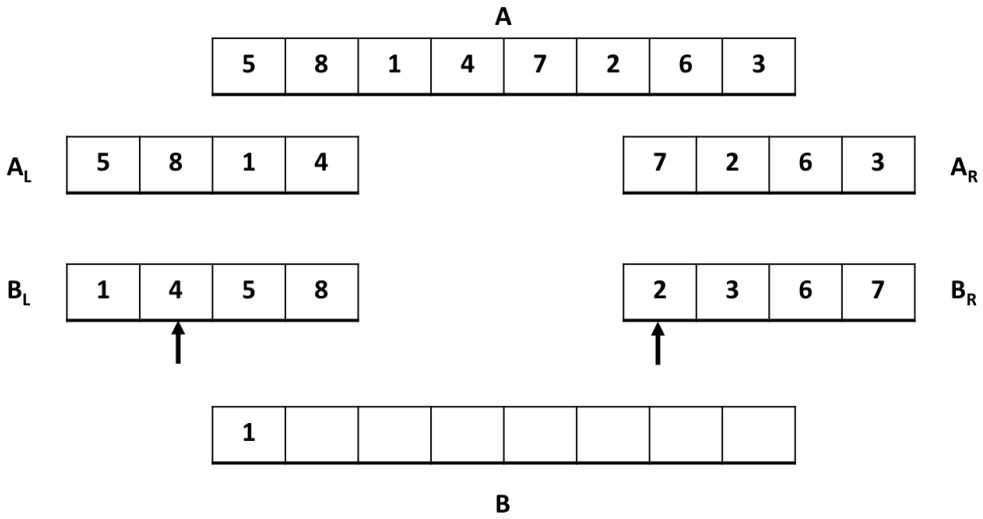
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

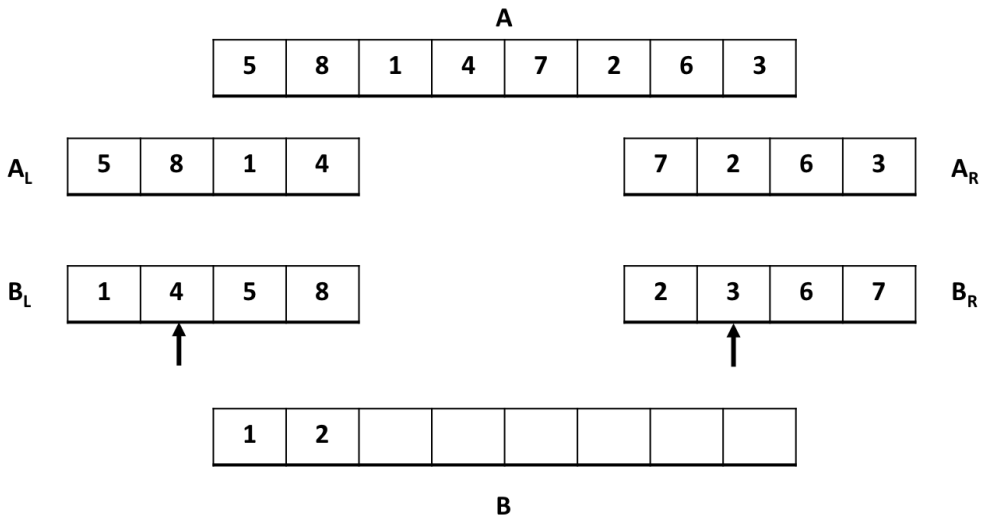
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

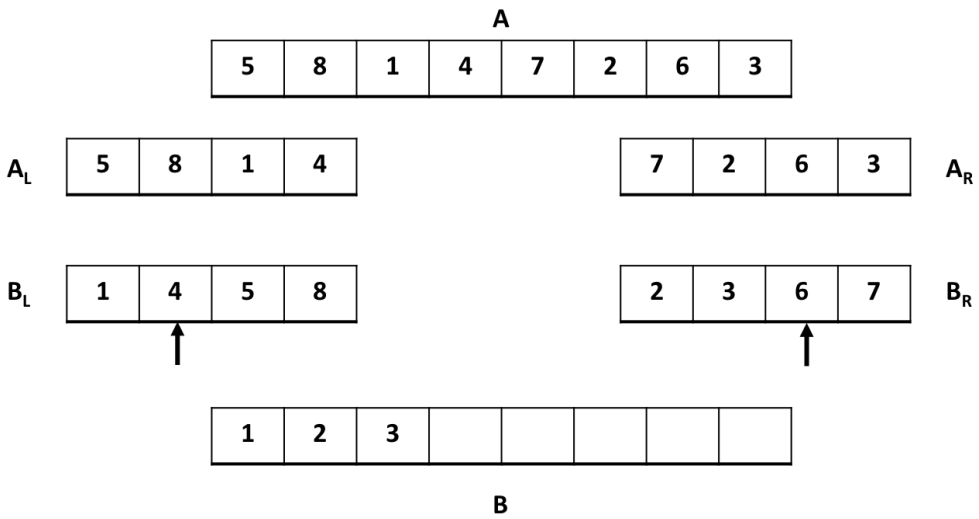
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

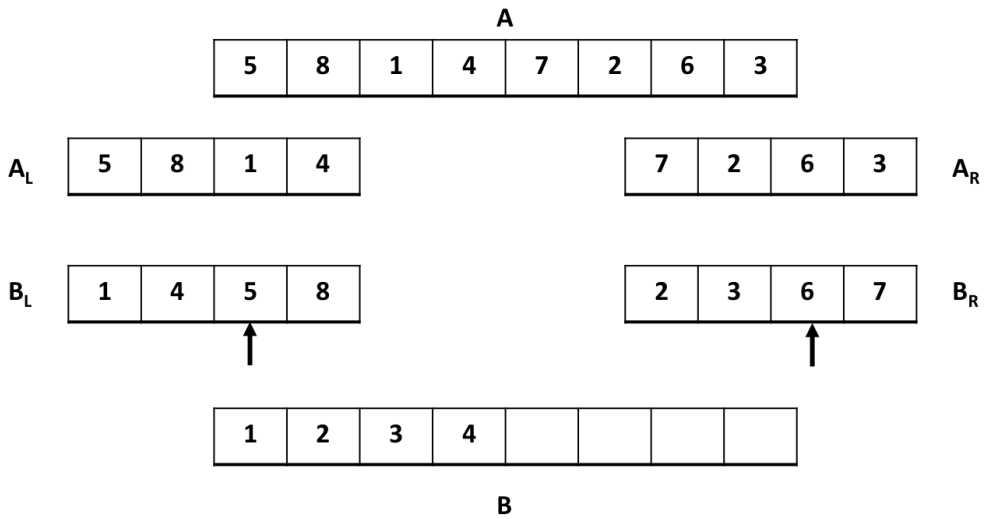
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

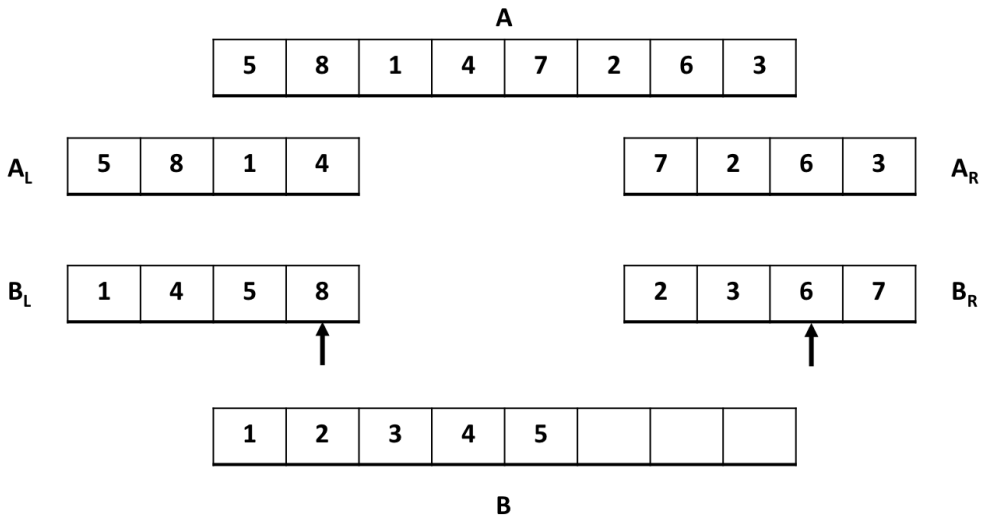
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

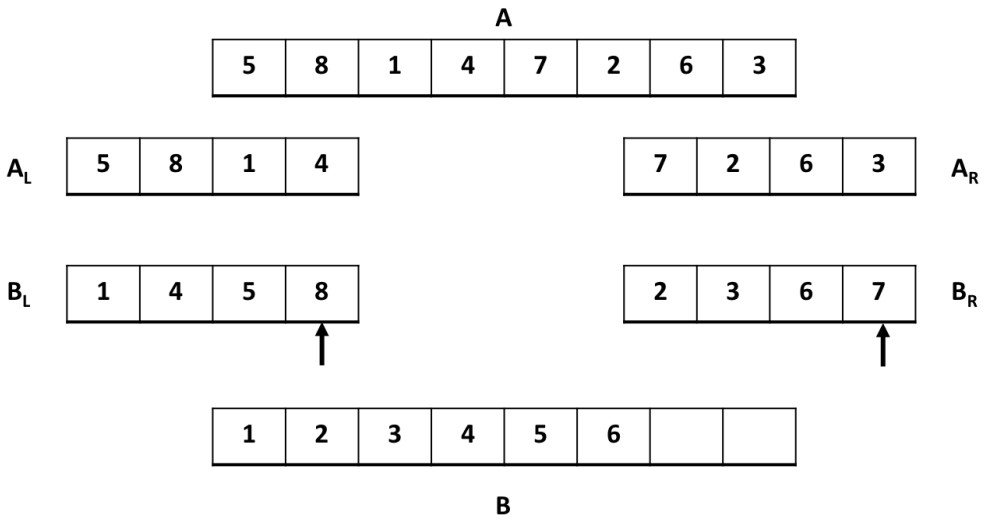
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

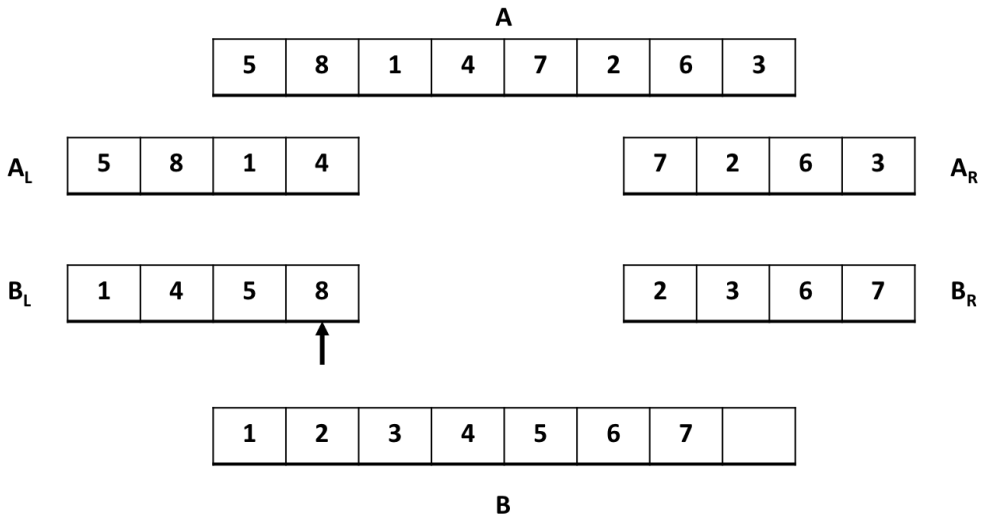
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

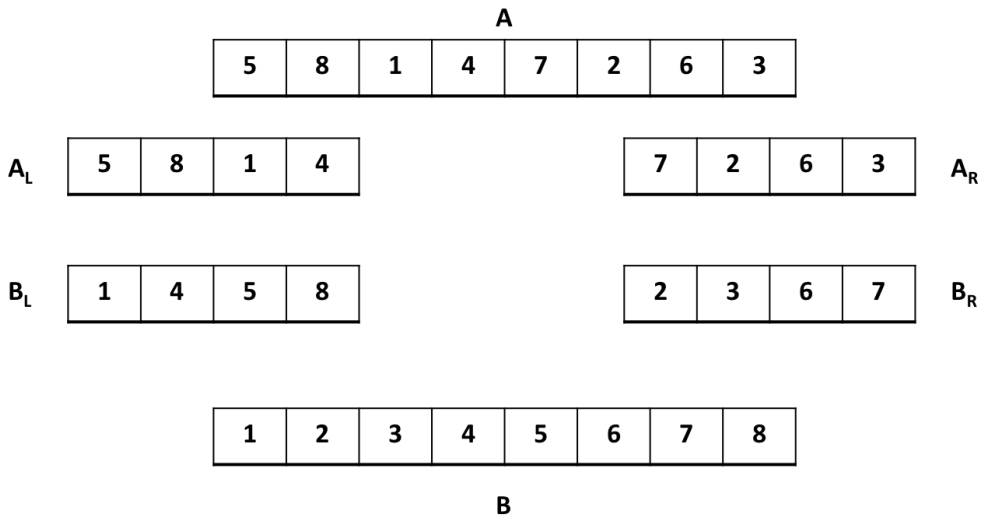
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort(`A`)`

- If ($|A| = 1$) return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort(`A_L`)`
- $B_R \leftarrow$ `MergeSort(`A_R`)`
- $B \leftarrow$ `Merge(`B_L, B_R`)`
- return(B)



Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- How do we argue correctness?

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- How do we argue correctness?
- Proof of correctness of Divide and Conquer algorithms are usually by induction.
  - Base case: This corresponds to the base cases of the algorithm. For the `MergeSort`, the base case is that the algorithm correctly sorts arrays of size 1.
  - Inductive step: In general, this corresponds to correctly combining the solutions of smaller subproblems. For `MergeSort`, this is just proving that the `Merge` routine works correctly. This may again be done using induction and is left as an exercise.

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- What is the running time of `MergeSort`?

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- Recurrence relation for running time: $T(n) \leq 2 \cdot T(n/2) + cn$ for all $n \geq 2$ and $T(1) \leq c$ for some constant c.
- Obtain the solution to the above recurrence relation by unrolling the recursion.

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- Recurrence relation for running time: $T(n) \leq 2 \cdot T(n/2) + cn$ for all $n \geq 2$ and $T(1) \leq c$ for some constant c.
- Obtain the solution to the above recurrence relation by unrolling the recursion. $T(n) = O(n \log n)$

- Consider the recurrence relation for the running time of the `MergeSort` algorithm:

$$T(n) \leq 2 \cdot T(n/2) + cn \text{ for all } n \geq 2 \; ; T(2) \leq c$$

- Another way to solve the recurrence relation is *substitution*:
  1. Guess the bound on $T(n)$, and
  2. Show that this bound holds using induction.
- Let our guess be $T(n) \leq cn \log n$ for all $n \geq 2$. We will now prove this by induction
- Base case: $T(n) \leq cn \log n$ when $n = 2$ since we are given that $T(2) \leq c$.
- Inductive step: Suppose the bound holds for $n = 2, ..., k-1$, we will show that the bound also holds for $n = k$.
  - We know $T(k) \leq 2T(k/2) + ck$.
  - So, using induction hypothesis, we get: $T(k) \leq 2c(k/2)\log(k/2) + ck = ck \log k$.

Data Structures

- How do Data Structures play a part in making computational tasks efficient?
  - We saw an example where organising and accessing data plays an important role in determining the efficiency of the computational task.

- How do Data Structures play a part in making computational tasks efficient?
  - We saw an example where organising and accessing data plays an important role in determining the efficiency of the computational task.
  - In certain computational tasks of limited nature where a specific way of organizing and accessing data makes sense. The nature of the computational task itself guides the kind of data structure that is most appropriate.

- How do Data Structures play a part in making computational tasks efficient?
  - We saw an example where organising and accessing data plays an important role in determining the efficiency of the computational task.
  - *In certain computational tasks of limited nature where a specific way of organizing and accessing data makes sense. The nature of the computational task itself guides the kind of data structure that is most appropriate.*
- Suppose you have to automate the queuing service at the local Doctor's office.
- The main requirement for such a service is **First In First Out** (FIFO in short).
  - As people come to the Doctor's office, they enter their names into the computer. The Doctor asks the computer to return the name of the next person as per the order in which they turned up.

Problem

Automate the queueing service at a Doctor's office. As people come to the Doctor's office, they enter their names into the computer. The Doctor asks the computer to return the name of the next person as per the order in which they turned up.

- The data structure that you design needs to support only the following two operations:
  1. `Enqueue(`*Name*`)`: Insert the name of the person.
  2. `Dequeue()`: Remove and return the name of the person who came first and has not been served yet.
- Such an *Abstract Data Type* (ADT in short) is called a Queue.
  - Abstract Data Type: A Mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations.

Problem

Automate the queueing service at a Doctor's office. As people come to the Doctor's office, they enter their names into the computer. The Doctor asks the computer to return the name of the next person as per the order in which they turned up.

- The data structure that you design needs to support only the following two operations:
  1. `Enqueue(`e`)`: Add element e to the back of the queue.
  2. `Dequeue()`: Remove and return the first element of the queue (or null if the queue is empty).
- Such an *Abstract Data Type* (ADT in short) is called a Queue.
- Question: Can you implement a queue using an array?

- Consider a sequence of function calls in some programming language:
  - $f(a)\{g(a); ...\}$
  - $g(a)\{h(a); ...\}$
  - $h(a)\{...\}$
- What happens when we make a call $f(10)$:
  - f makes a call to g: We need some data structure to save the state of the function f so that when the call to g returns, we should be able to execute the remaining instructions of f.
  - g makes a call to h: Similarly, we need to save the state of g to be able to run remaining instructions of g
- We need a data structure where we can store the state of a function and extract these saved states in a **Last In First Out** (LIFO) order.

- Consider a sequence of function calls in some programming language:
  - $f(a)\{g(a); ...\}$
  - $g(a)\{h(a); ...\}$
  - $h(a)\{...\}$
- What happens when we make a call $f(10)$:
  - f makes a call to g: We need some data structure to save the state of the function f so that when the call to g returns, we should be able to execute the remaining instructions of f.
  - g makes a call to h: Similarly, we need to save the state of g to be able to run remaining instructions of g
- We need a data structure where we can store the state of a function and extract these saved states in a **Last In First Out** (LIFO) order.
- Such a Data Structure needs to support the following two main operations:
  1. `Push(`e`)`: Add element e
  2. `Pop()`: Removes and returns the in LIFO order (or null if the stack is empty).

- We need a data structure where we can store the state of a function and extract these saved states in a **Last In First Out** (LIFO) order.
- Such a Data Structure needs to support the following two main operations:
  1. `Push(`e`)`: Add element e
  2. `Pop()`: Removes and returns the in LIFO order (or null if the stack is empty).
- Such an abstract data type is called a Stack.

- We need a data structure where we can store the state of a function and extract these saved states in a **Last In First Out** (LIFO) order.
- Such a Data Structure needs to support the following two main operations:
  1. `Push(`e`)`: Add element e
  2. `Pop()`: Removes and returns the in LIFO order (or null if the stack is empty).
- Such an abstract data type is called a Stack.
- Can you implement a stack using an array?

- We need a data structure where we can store the state of a function and extract these saved states in a **Last In First Out** (LIFO) order.
- Such a Data Structure needs to support the following two main operations:
  1. `Push(`e`)`: Add element e
  2. `Pop()`: Removes and returns the in LIFO order (or null if the stack is empty).
- Such an abstract data type is called a Stack.
- Can you implement a stack using an array? What is the running time for each operation?

- Can you implement a stack using an array? What is the running time for each operation?
- Can you implement a stack using a queue? What is the running time for each operation?

- Can you implement a stack using an array? What is the running time for each operation?
- Can you implement a stack using a queue? What is the running time for each operation?
- Can you implement a queue using a stack?

- Can you implement a stack using an array? What is the running time for each operation?
- Can you implement a stack using a queue? What is the running time for each operation?
- Can you implement a queue using a stack?
- Can you implement a queue using two stacks?

- Can you implement a stack using an array? What is the running time for each operation?
- Can you implement a stack using a queue? What is the running time for each operation?
- Can you implement a queue using a stack?
- Can you implement a queue using two stacks? What is the running time for each operation?

End