

COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Data Structures

Queue and 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? What is the running time for each operation?

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B .
- $\text{Enqueue}(e)$: $\text{Push}_A(e)$

Algorithm

$\text{Dequeue}()$

- If (A and B are empty)
 - return($null$)
- If (B is not empty)
 - return($\text{Pop}_B()$)
- while(A is not empty)
 - $\text{Push}_B(\text{Pop}_A())$
- return($\text{Pop}_B()$)

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B .
- $\text{Enqueue}(e)$: $\text{Push}_A(e)$

Algorithm

$\text{Dequeue}()$

- If (A and B are empty)
 - return(null)
- If (B is not empty)
 - return($\text{Pop}_B()$)
- while(A is not empty)
 - $\text{Push}_B(\text{Pop}_A())$
- return($\text{Pop}_B()$)

- What is the running time of each of the basic operations:
 - $\text{Enqueue}(e)$:
 - $\text{Dequeue}()$:

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B .
- $\text{Enqueue}(e)$: $\text{Push}_A(e)$

Algorithm

$\text{Dequeue}()$

- If (A and B are empty)
 - return($null$)
- If (B is not empty)
 - return($\text{Pop}_B()$)
- while(A is not empty)
 - $\text{Push}_B(\text{Pop}_A())$
- return($\text{Pop}_B()$)

- What is the running time of each of the basic operations:
 - $\text{Enqueue}(e)$: $O(1)$
 - $\text{Dequeue}()$: $O(n)$

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B .
- $\text{Enqueue}(e)$: $\text{Push}_A(e)$

Algorithm

$\text{Dequeue}()$

- If (A and B are empty)
 - return(null)
- If (B is not empty)
 - return($\text{Pop}_B()$)
- while(A is not empty)
 - $\text{Push}_B(\text{Pop}_A())$
- return($\text{Pop}_B()$)

- What is the running time of each of the basic operations:
 - $\text{Enqueue}(e)$: $O(1)$
 - $\text{Dequeue}()$: $O(n)$
- Comment: It is very pessimistic to say that the running time of the $\text{Dequeue}()$ operation is $O(n)$ (even though correct).
- Is there a better way to analyse the running time in such scenarios where an operation is costly **only sometimes**?

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B .
- $\text{Enqueue}(e)$: $\text{Push}_A(e)$

Algorithm

$\text{Dequeue}()$

- If (A and B are empty)
 - return($null$)
- If (B is not empty)
 - return($\text{Pop}_B()$)
- while(A is not empty)
 - $\text{Push}_B(\text{Pop}_A())$
- return($\text{Pop}_B()$)

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of $\text{Dequeue}()$ operation?

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of `Dequeue()` operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time.

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of Dequeue() operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. **But, then the next $|A|$ dequeue operations will take $O(1)$ time.**

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of Dequeue() operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. **But, then the next $|A|$ dequeue operations will take $O(1)$ time.**
- We can compute the total cost by considering the **cost per element** of the queue and then sum over all elements.

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of `Dequeue()` operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. **But, then the next $|A|$ dequeue operations will take $O(1)$ time.**
- We can compute the total cost by considering the **cost per element** of the queue and then sum over all elements.
- What is the cost associated with each element?
 - ① The element is pushed into Stack A .
 - ② The element (at some point of time) needs to be moved from Stack A to Stack B .
 - ③ The element is finally popped out from B .

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of `Dequeue()` operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. **But, then the next $|A|$ dequeue operations will take $O(1)$ time.**
- We can compute the total cost by considering the **cost per element** of the queue and then sum over all elements.
- What is the cost associated with each element? **4 operations**
 - ① The element is pushed into Stack A . This costs 1 operation.
 - ② The element (at some point of time) needs to be moved from Stack A to Stack B . This costs 2 operations.
 - ③ The element is finally popped out from B . This costs 1 operation.

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of Dequeue() operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. **But, then the next $|A|$ dequeue operations will take $O(1)$ time.**
- We can compute the total cost by considering the **cost per element** of the queue and then sum over all elements.
- What is the cost of dequeue associated with each element? **4 operations**
 - ① The element is pushed into Stack A . This costs 1 operation.
 - ② The element (at some point of time) needs to be moved from Stack A to Stack B . This costs 2 operations.
 - ③ The element is finally popped out from B . This costs 1 operation.
- So, $O(n)$ operations will be performed in total over a series of n operations.

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of Dequeue() operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. But, then the next $|A|$ dequeue operations will take $O(1)$ time.
- We can compute the total cost by considering the cost per element of the queue and then sum over all elements.
- What is the cost of dequeue associated with each element? 4 operations
 - ① The element is pushed into Stack A . This costs 1 operation.
 - ② The element (at some point of time) needs to be moved from Stack A to Stack B . This costs 2 operations.
 - ③ The element is finally popped out from B . This costs 1 operation.
- So, $O(n)$ operations will be performed in total over a series of n operations.
- So, the amortized running time for the operations are:
 - Enqueue(e): $O(1)$
 - Dequeue(): $O(1)$

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Suppose a series of n operations are performed on the queue. What is the average running time of Dequeue() operation?
 - When B is not empty, then the operation takes $O(1)$ times.
 - When B is empty, then one has to pop all elements from A and push them into B . This would take $O(|A|)$ time. **But, then the next $|A|$ dequeue operations will take $O(1)$ time.**
- Another way of viewing this analysis is through of concept of *budgeting*.
- Here, we will argue that $4n$ coins are enough to fund n Queue operation where you pay one coin for every simple operation performed in the implementation (that is, on the stack(s)).

Data Structures

Digression: Queue and Stack $\rightarrow$ Amortized Analysis

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B .
- $\text{Enqueue}(e)$: $\text{Push}_A(e)$

Algorithm

$\text{Dequeue}()$

- If (A and B are empty)
 - return(null)
- If (B is not empty)
 - return($\text{Pop}_B()$)
- while(A is not empty)
 - $\text{Push}_B(\text{Pop}_A())$
- return($\text{Pop}_B()$)

- Amortized analysis: Per-operation running time averaged over a series of operations.
- So, the **amortized running time** for the operations are:
 - $\text{Enqueue}(e)$: $O(1)$
 - $\text{Dequeue}()$: $O(1)$

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Let us see another example of Amortized analysis in Data Structures.
- Arrays are most basic data structures where elements can be accessed using *indices*. Contiguous memory locations are used for arrays.
- One common issue while using Arrays is that the array size is fixed once defined and if the array becomes full then there is no way to handle the **overflow**.

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Let us see another example of Amortized analysis in Data Structures.
- Arrays are most basic data structures where elements can be accessed using *indices*. Contiguous memory locations are used for arrays.
- One common issue while using Arrays is that the array size is fixed once defined and if the array becomes full then there is no way to handle the **overflow**.
- **Dynamic Arrays** are arrays that are **resizable** and allows to accommodate arbitrary number of elements.
- Such arrays are dynamic in the sense that the array “dynamically adjusts itself in case of overflows”.

Data Structures

Digression: Queue and Stack → Amortized Analysis

- Amortized analysis: Per-operation running time averaged over a series of operations.
- Let us see another example of Amortized analysis in Data Structures.
- Arrays are most basic data structures where elements can be accessed using *indices*. Contiguous memory locations are used for arrays.
- One common issue while using Arrays is that the array size is fixed once defined and if the array becomes full then there is no way to handle the **overflow**.
- **Dynamic Arrays** are arrays that are **resizable** and allows to accommodate arbitrary number of elements.
- Such arrays are dynamic in the sense that the array “dynamically adjusts itself in case of overflows”.
- Can you implement a dynamic arrays using an regular arrays?

Data Structures

Digression: Queue and Stack → Amortized Analysis

Problem

Implement Dynamic Arrays using regular Arrays.

- Initialisation: Create an array (say A) of some constant size.
- Overflow: Every time the array *overflows*, do:
 - Create an array B double the size of the current array A (i.e., $|B| = 2|A|$)
 - Copy all the elements of A into B
 - Rename B as A

Data Structures

Digression: Queue and Stack → Amortized Analysis

Problem

Implement Dynamic Arrays using regular Arrays.

- Initialisation: Create an array (say A) of some constant size.
- Overflow: Every time the array *overflows*, do:
 - Create an array B double the size of the current array A (i.e., $|B| = 2|A|$)
 - Copy all the elements of A into B
 - Rename B as A
- What is the running time of the insert operation?

Data Structures

Digression: Queue and Stack → Amortized Analysis

Problem

Implement Dynamic Arrays using regular Arrays.

- Initialisation: Create an array (say A) of some constant size.
- Overflow: Every time the array *overflows*, do:
 - Create an array B double the size of the current array A (i.e., $|B| = 2|A|$)
 - Copy all the elements of A into B
 - Rename B as A
- What is the running time of the insert operation? $O(n)$

Data Structures

Digression: Queue and Stack → Amortized Analysis

Problem

Implement Dynamic Arrays using regular Arrays.

- Initialisation: Create an array (say A) of some constant size.
- Overflow: Every time the array *overflows*, do:
 - Create an array B double the size of the current array A (i.e, $|B| = 2|A|$)
 - Copy all the elements of A into B
 - Rename B as A
- What is the running time of the insert operation? $O(n)$
- What is the **Amortized** running time of insert operation?

End