COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

- 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?

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B.
- `Enqueue`(e): `Push`$_A(e)$

Algorithm

`Dequeue()`
- If (A and B are empty)
  - return(*null*)
- If (B is not empty)
  - return($\texttt{Pop}_\texttt{B}()$)
- while(A is not empty)
  - $\texttt{Push}_\texttt{B}(\texttt{Pop}_\texttt{A}())$
- return($\texttt{Pop}_\texttt{B}()$)

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B.
- `Enqueue(`e`)`: `Push`$_A(e)$

Algorithm

`Deque()`
- If (A and B are empty)
  - return(*null*)
- If (B is not empty)
  - return($\texttt{Pop}_{\texttt{B}}()$)
- while(A is not empty)
  - $\texttt{Push}_{\texttt{B}}(\texttt{Pop}_{\texttt{A}}())$
- return($\texttt{Pop}_{\texttt{B}}()$)

- What is the running time of each of the basic operations:
  - `Enqueue(`e`)`:
  - `Dequeue()`:

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B.
- `Enqueue`(e): `Push`$_A(e)$

Algorithm

`Dequeue()`
- If (A and B are empty)
  - return(*null*)
- If (B is not empty)
  - return(`Pop`$_\texttt{B}$())
- while(A is not empty)
  - `Push`$_\texttt{B}$(`Pop`$_\texttt{A}$())
- return(`Pop`$_\texttt{B}$())

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

Problem

Implement a Queue using two stacks.

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

Algorithm

`Deque()`
- If (A and B are empty)
  - return($null$)
- If (B is not empty)
  - return($\texttt{Pop}_\texttt{B}()$)
- while(A is not empty)
  - $\texttt{Push}_\texttt{B}(\texttt{Pop}_\texttt{A}())$
- return($\texttt{Pop}_\texttt{B}()$)

- What is the running time of each of the basic operations:
  - `Enqueue`(e): $O(1)$
  - `Dequeue()`: $O(n)$
- Comment: It is very pessimistic to say that the running time of the `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?

Problem

Implement a Queue using two stacks.

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

Algorithm

`Dequeue()`
- If (A and B are empty)
  - return(*null*)
- If (B is not empty)
  - return($\texttt{Pop}_\texttt{B}$())
- while(A is not empty)
  - $\texttt{Push}_\texttt{B}$($\texttt{Pop}_\texttt{A}$())
- return($\texttt{Pop}_\texttt{B}$())

- 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 `Dequeue()` operation?

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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()`: $O(1)$

- 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)).

Problem

Implement a Queue using two stacks.

- Let the two stacks be A and B.
- `Enqueue`(e): `Push`$_A(e)$

Algorithm

`Dequeue()`
- If (A and B are empty)
  - return(*null*)
- If (B is not empty)
  - return($\texttt{Pop}_\texttt{B}()$)
- while(A is not empty)
  - $\texttt{Push}_\texttt{B}(\texttt{Pop}_\texttt{A}())$
- return($\texttt{Pop}_\texttt{B}()$)

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

- 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.

- 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".

- 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?

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

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 insert operation?

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 insert operation? $O(n)$

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 insert operation? $O(n)$
- What is the Amortized running time of insert operation?

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 insert operation? $O(n)$
- What is the Amortized running time of insert operation?
  - Suppose starting from the empty array (of size 1) one performs n insert operation in a sequence. What is the total running time?

- What is the Amortized running time of insert operation?
  - Suppose starting from the empty array (of size 1) one performs n insert operation in a sequence. What is the total running time?

#	Operation	Work done	# basic ops.
1	Insert(8)	[8]	1
2	Insert(5)	[8] → [8, 5]	(1 + 1)
3	Insert(0)	[8, 5] → [8, 5, 0,]	(2 + 1)
4	Insert(2)	[8, 5, 0,] → [8, 5, 0, 2]	1
5	Insert(3)	[8, 5, 0, 2] → [8, 5, 0, 2, 3, , ,]	(4 + 1)
6	Insert(9)	[8, 5, 0, 2, 3, 9, ,]	1
7	Insert(4)	[8, 5, 0, 2, 3, 9, 4,]	1
8	Insert(6)	[8, 5, 0, 2, 3, 9, 4, 6]	1

- What is the Amortized running time of insert operation?
  - Suppose starting from the empty array (of size 1) one performs n insert operation in a sequence. What is the total running time?
  - If $2^k \leq n < 2^{k+1}$, then

$$\begin{aligned}\#\text{ Basic ops.} &= n + (1 + 2 + 2^2 + ... + 2^k) \\ &= n + 2^{k+1} - 1 \\ &\leq 3n - 1 \\ &= O(n)\end{aligned}$$

- What is the Amortized running time of insert operation?
  - Suppose starting from the empty array (of size 1) one performs n insert operation in a sequence. What is the total running time?
  - If $2^k \leq n < 2^{k+1}$, then

$$\begin{aligned}
\#\text{ Basic ops.} &= n + (1 + 2 + 2^2 + ... + 2^k) \\
&= n + 2^{k+1} - 1 \\
&\leq 3n - 1 \\
&= O(n)
\end{aligned}$$

- So, the Amortized running time for the insert operation is $O(1)$.

- What is the Amortized running time of insert operation? $O(1)$
- Budgeting:
  - The cost of copying can be charged to the new cell locations that are created.
  - So, the total number of coins required will be n (for inserts) and at most $2n$ (for copies).

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 insert operation? $O(n)$
- What is the Amortized running time of insert operation? $O(1)$

- One issue with Arrays is that they are not re-sizeable.
- If the only operations that need to be supported are insert and search, then Dynamic Arrays solve the issue of overflow.
- Suppose we also need to support deletion of a particular element or insertion of an element in the middle of the array.
- These operations are costly on Arrays since the elements need to be "shifted" to maintain contiguity.
- One data structure that does not have this issue is Linked List.

- Linked List: A collection of nodes with linear ordering defined on them.
    - Each node holds an element and points to the next node in the order.
    - The first node in the ordering is called the head and the last is called the tail.
    - The tail points to a null reference.
    - The data structure is accessed using a reference to the head node.

- Linked List: A collection of nodes with linear ordering defined on them.
  - Each node holds an element and points to the next node in the order.
  - The first node in the ordering is called the head and the last is called the tail.
  - The tail points to a null reference.
  - The data structure is accessed using a reference to the head node.

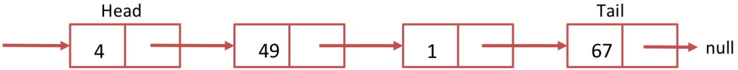

Figure : Visual representation of a Linked List

- Linked List: A collection of nodes with linear ordering defined on them.
  - Each node holds an element and points to the next node in the order.
  - The first node in the ordering is called the head and the last is called the tail.
  - The tail points to a null reference.
  - The data structure is accessed using a reference to the head node.
- Advantages of linked list:
  - The size of the data structure is roughly equal to the size of the elements that need to be stored. So, it is space-efficient.
  - The data structure is resizable.
  - "Shifting" not required as in the case of Arrays.

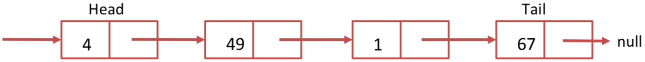

Figure : Visual representation of a Linked List

- Linked List: A collection of nodes with linear ordering defined on them.
  - Each node holds an element and points to the next node in the order.
  - The first node in the ordering is called the head and the last is called the tail.
  - The tail points to a null reference.
  - The data structure is accessed using a reference to the head node.
- Give the mechanism for performing the following operations along with the running time:
  - Add an element at the beginning of the list:
  - Add an element at the end of the list:
  - Delete a particular node (given its reference):
  - Delete the first node containing element e:
  - Search element e in the linked list:
  - Remove the first element of the list:

End