**Data Structures**

**What is a Data Structure?**

Data Structure is a way to organize data in such a way that enables it to be processed in an efficient time.

**Different Algorithm Runtime Complexities**

|  |  |  |  |
| --- | --- | --- | --- |
| **SNo** | **Time Complexity** | **Name** | **Example** |
| **1** | **O(1)** | **Constant** | Adding an element at front of LinkedList |
| **2** | **O(log n)** | **Logarithmic** | Finding an element in Sorted Array |
| **3** | **O(n)** | **Linear** | Finding an element in Unsorted Array |
| **4** | **O(n log n)** | **Linear Logarithmic** | Merge Sort |
| **5** | **O(n2)** | **Quadratic** | Shortest Path between 2 nodes in a Graph |
| **6** | **O(n3)** | **Cubic** | Matrix Multiplication |
| **7** | **O(2n)** | **Exponential** | Tower of Hanoi Problem |

**Below are different types of data structures**

1. Physical Data Structures
   1. Array
   2. Linked List
2. Logical Data Structures
   1. Tree
      1. Binary Tree
      2. BST
      3. Heap
      4. AVL Tree
      5. Trie
   2. Hashing
   3. Stack
   4. Queue
   5. Graph

**Array:**

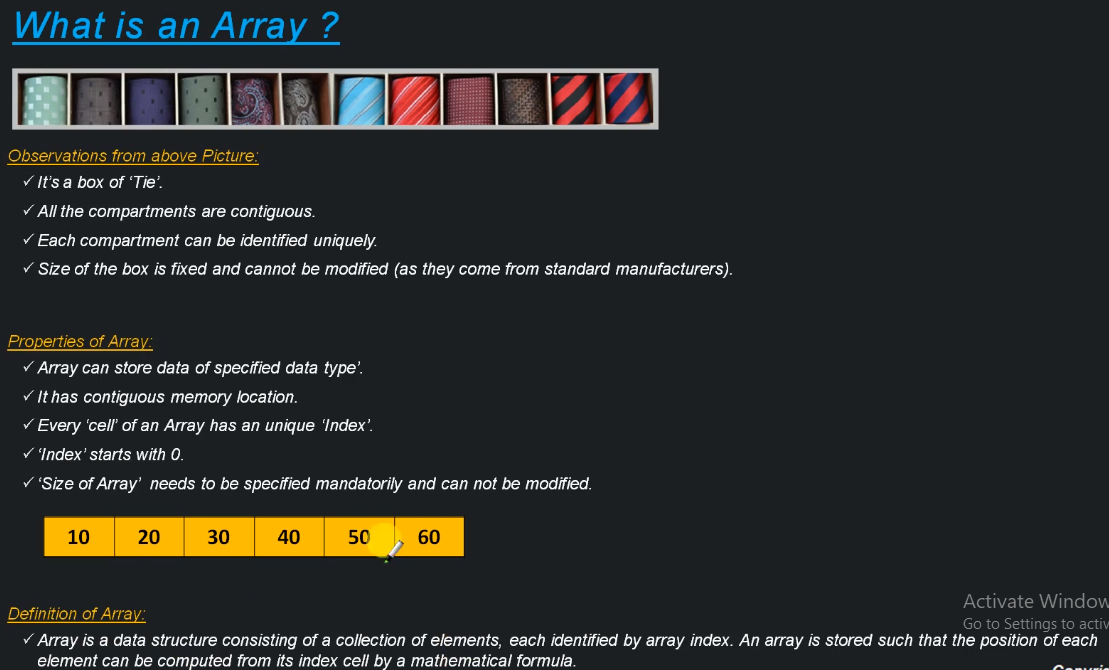
Array is a Data Structure which accepts homogenous data elements which can be stored index based which starts with 0.

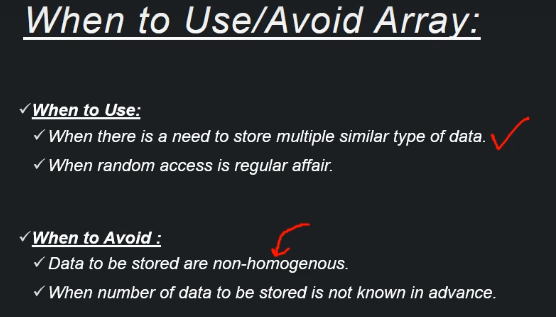
Each element of an array is of same type and same size.

Since the array index is 0 based, it’s easy to get the element from 0th index which is same as the array address.

To get the address of the a specific index, it can be evaluated easily as array address + size of a value \* index of the value

The best example of an array is a Tie box which container multiple compartments for each tie with similar size. It’s used in Hash tables to store values based on the hash value





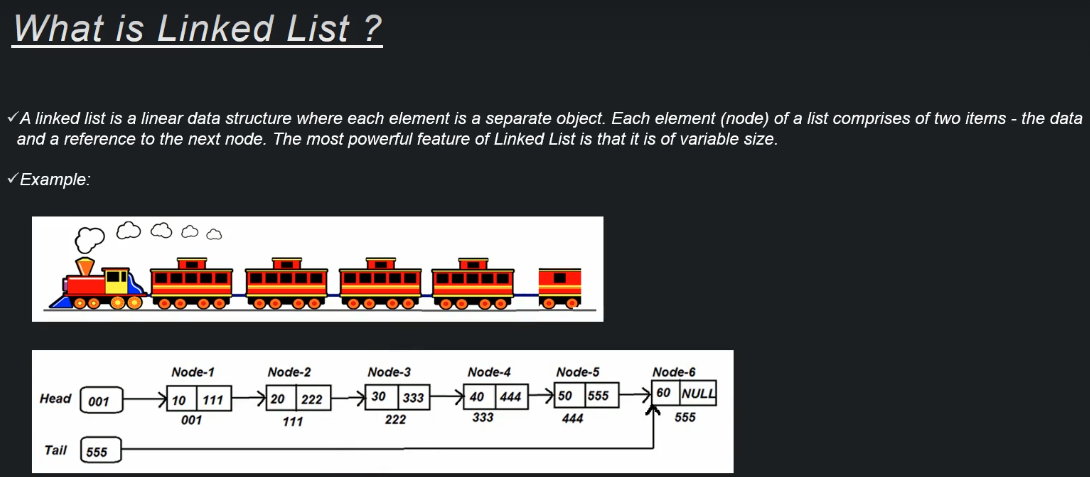
Below are the complexities involved with array operations.

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Retrieve with index | O(1) |
| Retrieve without index | O(n) |
| Add an element to full array | O(n) |
| Add an element to end of an array(has space) | O(1) |
| Insert or Update an element at a specific index | O(1) |
| Delete an element by setting it to null | O(1) |
| Delete an element by shifting elements | O(n) |

**Linked List:**

A Linked list is a linear data structure where each element is a separate object. Each element (node) of a list comprises of two items, the data and the reference to the next node. The most powerful feature of a Linked List is that it is of variable size.

Example is a Train, which always starts with an Engine and ends with a guard room, and contains multiple compartments in between. Engine -> Compartments -> Guard Room are linked together, any no. of compartments can be added or removed easily based on the needs. The couplers are the ones which joins them all.



Linked List vs Array

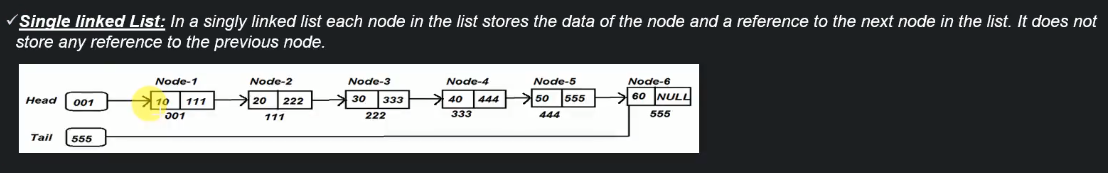
|  |  |  |
| --- | --- | --- |
| **Property** | **Array** | **Linked List** |
| Type of Data | Homogenous | Heterogeneous |
| Size | Fixed | Variable |
| Random Access (Ex: 3rd index) | Easy | Difficult |

Different Components in Linked List:

1. Node – Contains data and reference to next node.
2. Head – Contains the reference to First node in the Linked List.
3. Tail – Contains the reference to Last node in the Linked List. This is useful when we want to add a new Node at the end of last node.

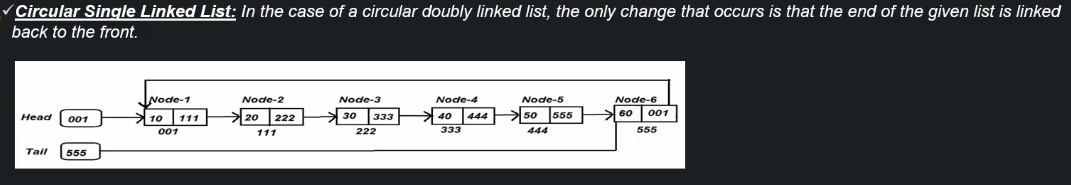
Types of Linked List:

1. Single Linked List



It’s the most basic form of Linked List which gives the flexibility to add/ remove nodes at runtime.

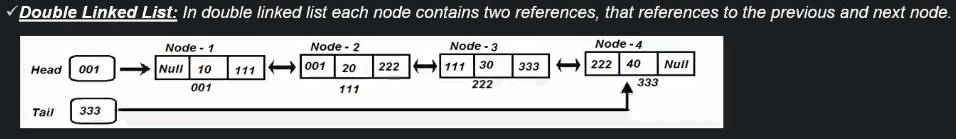
1. Circular Single Linked List



It’s useful when we want to loop through the list indefinitely until the list exists.

This is useful to implement multiplayer board game where if there are 4players, after completing the turns from player 1 to player 4, it should again start with player1, where player 4 must have the reference to player 1.

1. Double Linked List



It’s useful to move in both directions depending on the requirement.

Example is the music player which has next and previous buttons.

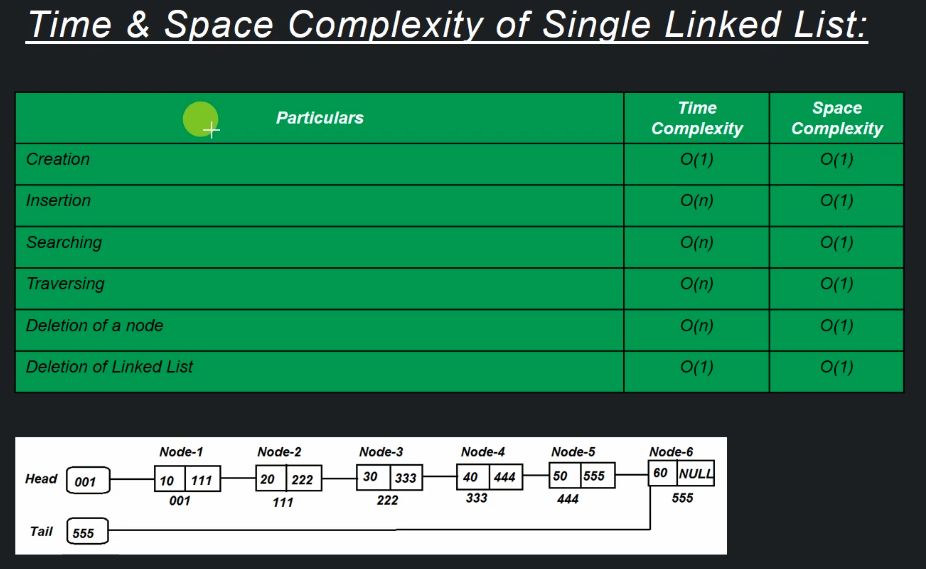
1. Circular Double Linked List

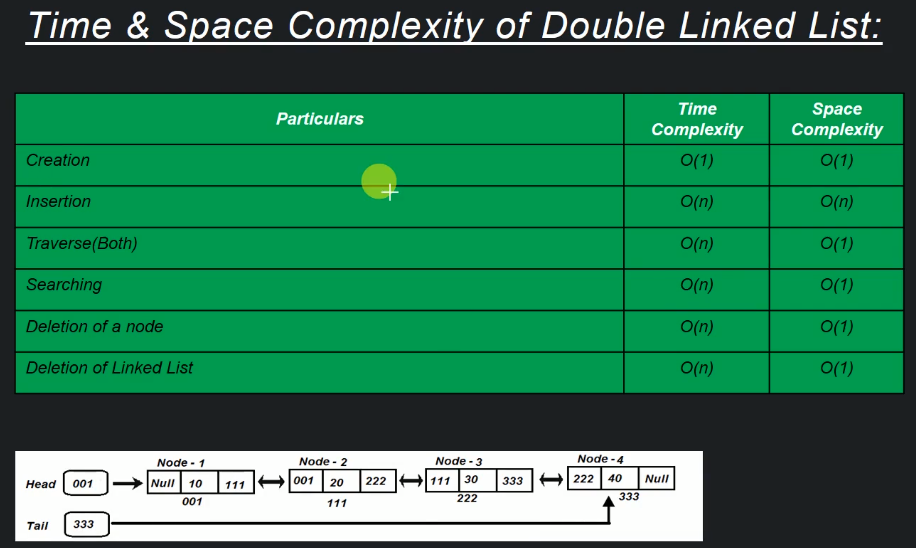


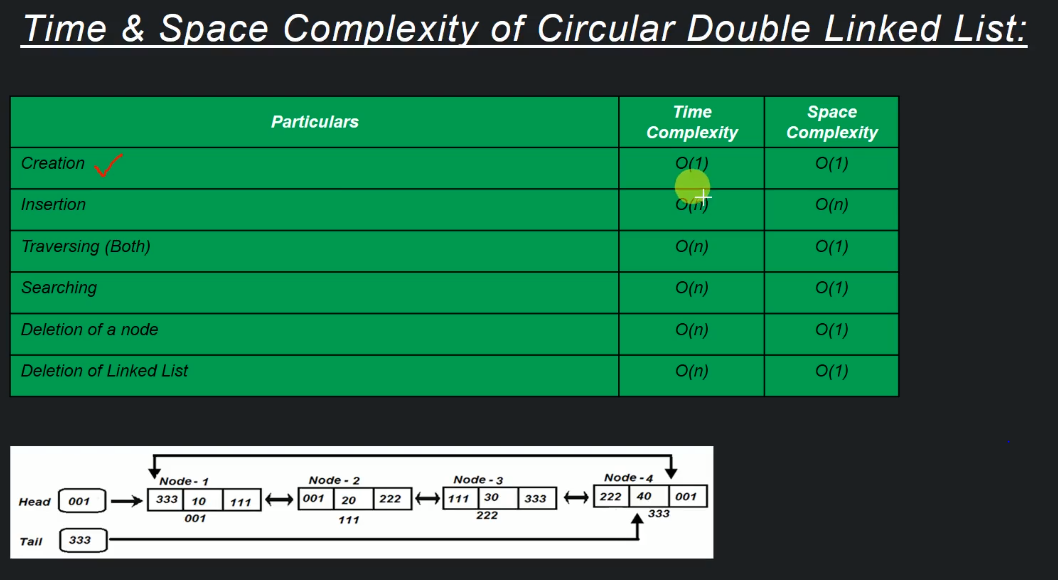
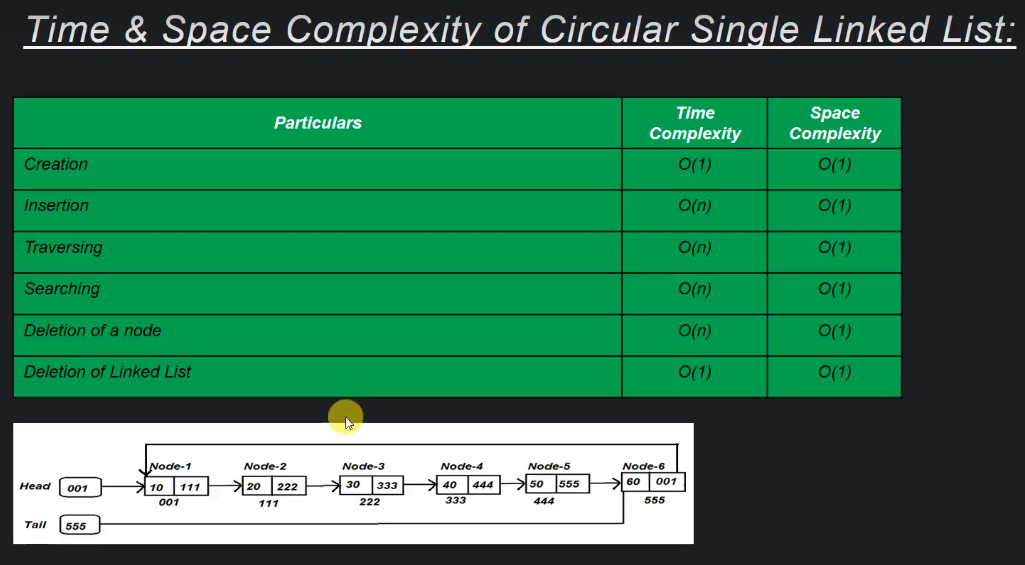
It’s useful when we want to loop through the list indefinitely until the list exist. We also want both forward and backward features

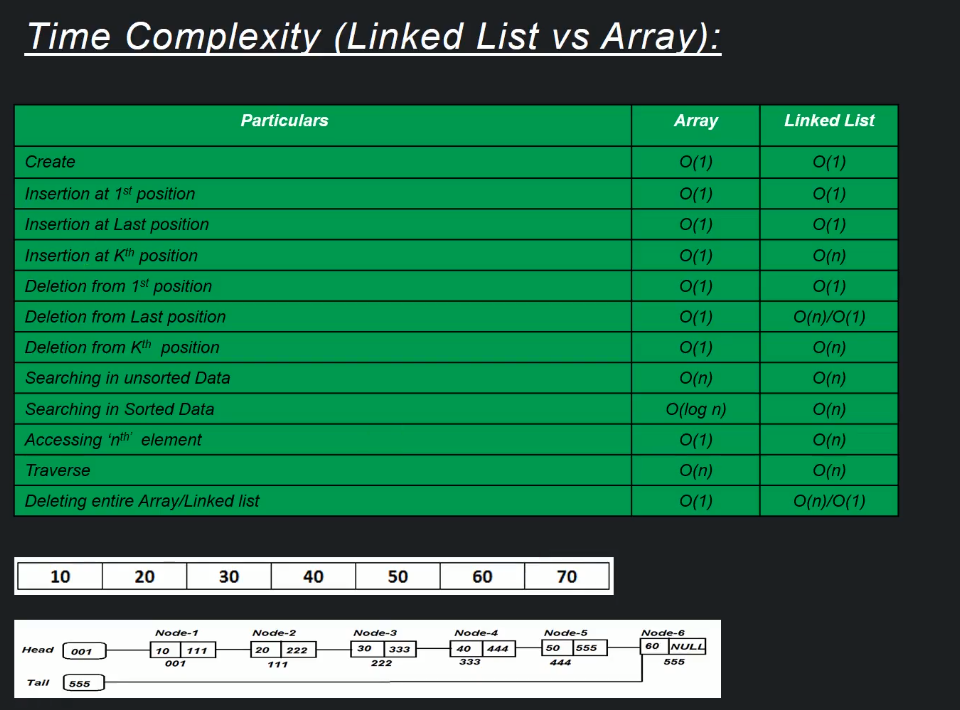
Example is Alt + Tab functionality where can traverse through back and forth to open a required window.

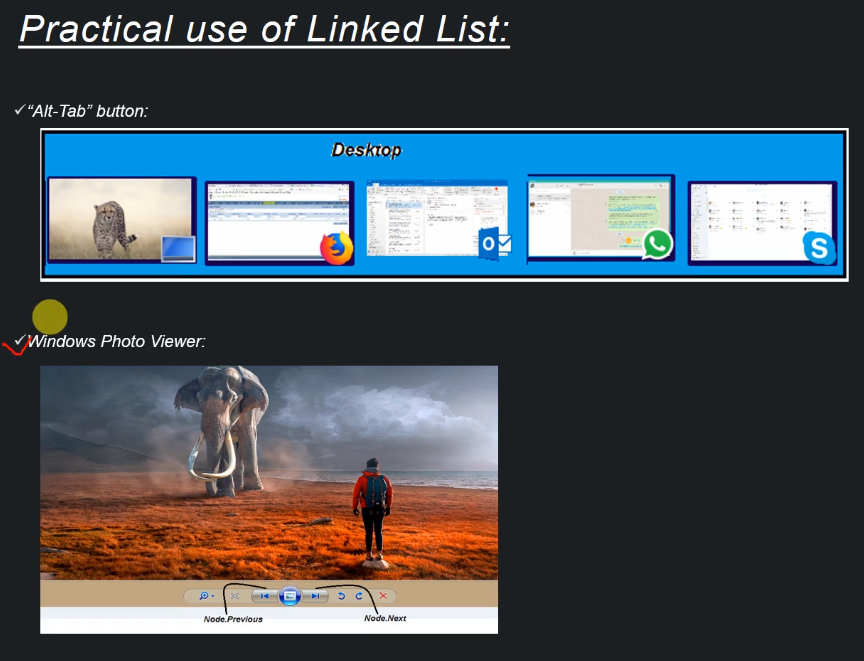
Time complexities





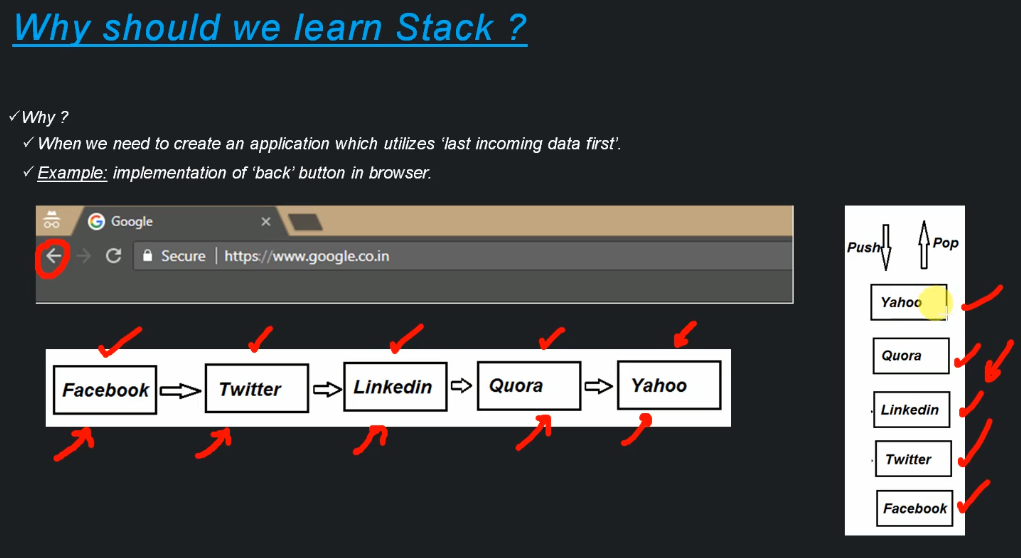




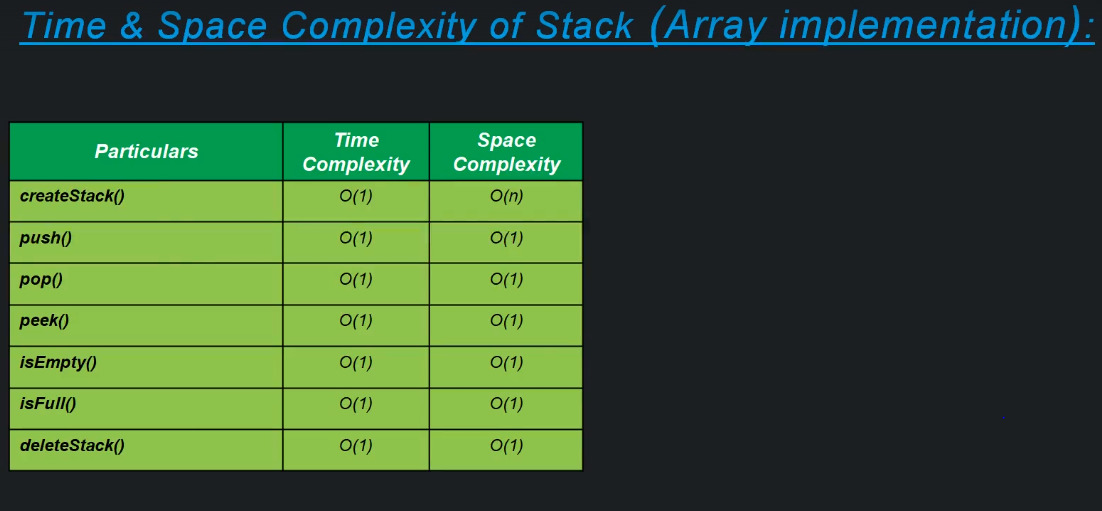


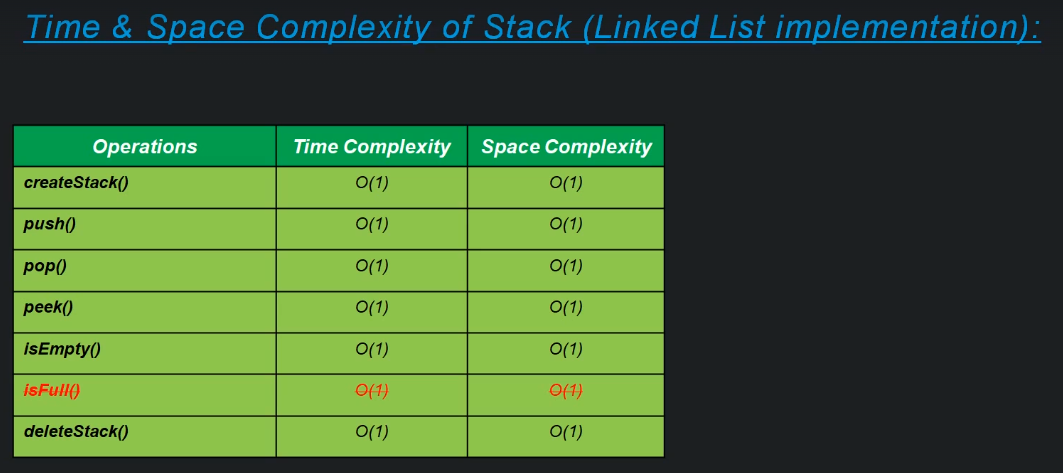
**Stack**

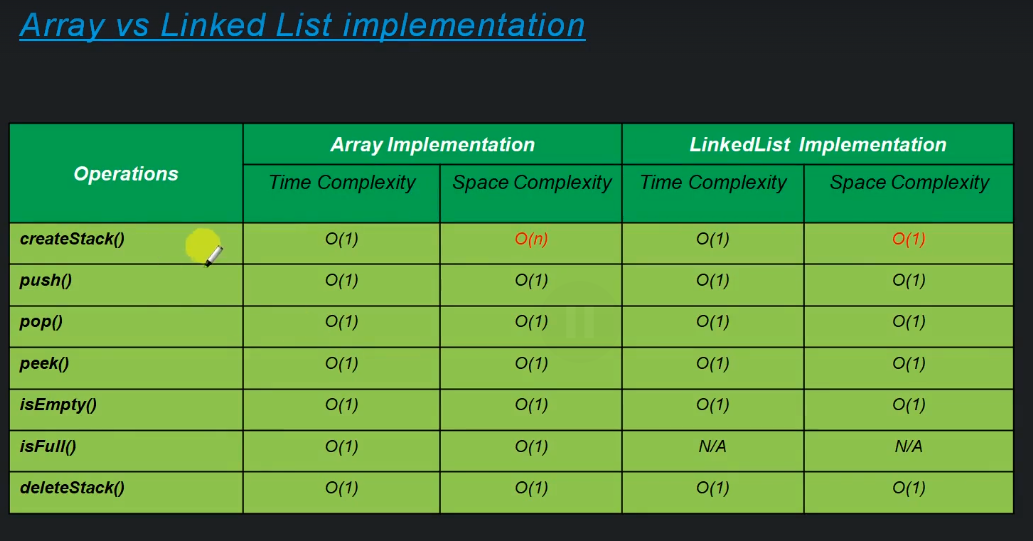


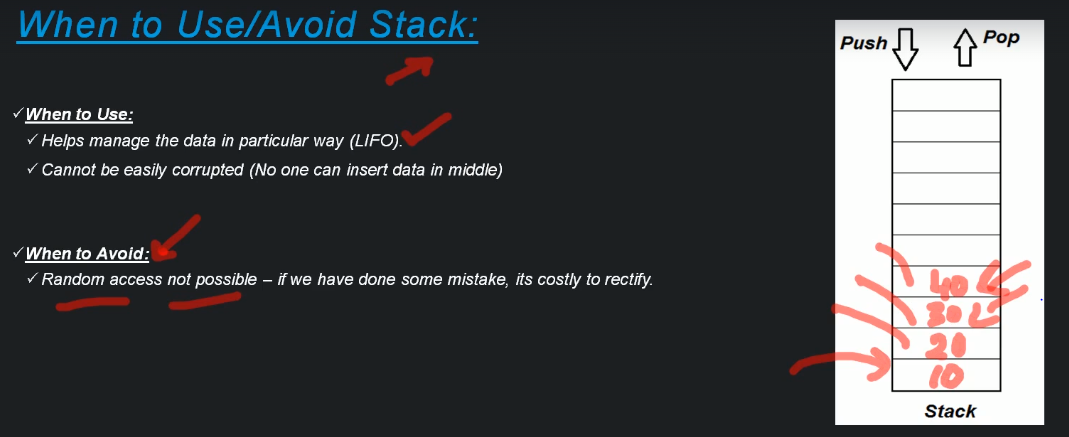


Time & Space Complexity:









**Queue:**

