Data Structures

https://www.javatpoint.com/jee-or-j2ee-design-patterns

java fork-join

java executor service

YouTube = [Telusko](https://www.youtube.com/channel/UC59K-uG2A5ogwIrHw4bmlEg) Complete playlist of Data Structure Using Java : [https://goo.gl/3eQAYB](https://www.youtube.com/redirect?redir_token=HCVlBGYdvnG16bo69HIcmunU9GR8MTU1NjI1NzgwOEAxNTU2MTcxNDA4&q=https%3A%2F%2Fgoo.gl%2F3eQAYB&event=video_description&v=M4Ql9DbKO6k)

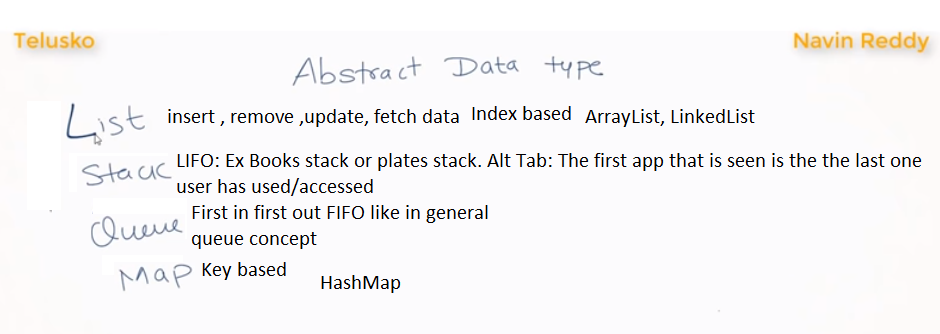
**Why Data Structures?**

* Data can be simple or complex.
* We need to store data such that it is easy to fetch it, to process it and store it again some where.
* It is about how can we structure your data so that we can store it and use it efficiently.
* list
* set
* map
* tree
* queue
* stack
* heap

Arrays: continuous allocation of memory

**ABSTRACT DATATYPE:**

Basically this means it is concept associated. In different programming languages they are implemented differently but the concept remains the **same**.Ex LinkedList and ArrayList the way they are implemented is different but the concept is the SAME.



**Arrays**

* Collection of values/data accessed using one variable
* The variable is declared /associated using array with []
* Each element is associated to an address.
* Array will give you a **contiguous** location (sharing a common border; touching. Or next or together in sequence.) ie if the first element is 104 the next element is next to it. There will **NOT be any gap** in between
* Elements are accessed using indices.
* **Advantages**:

We can store multiple values

We can access/fetch it fast using its index

* **Disadvantages**

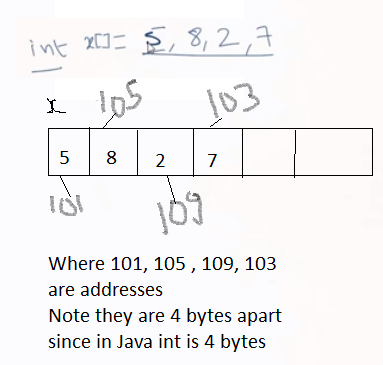
We cannot change the size of the array once the array is declared.

We cannot expand (Add elements) or shrink (delete, remove)

If we create an array and have some null values there is a wastage of memory

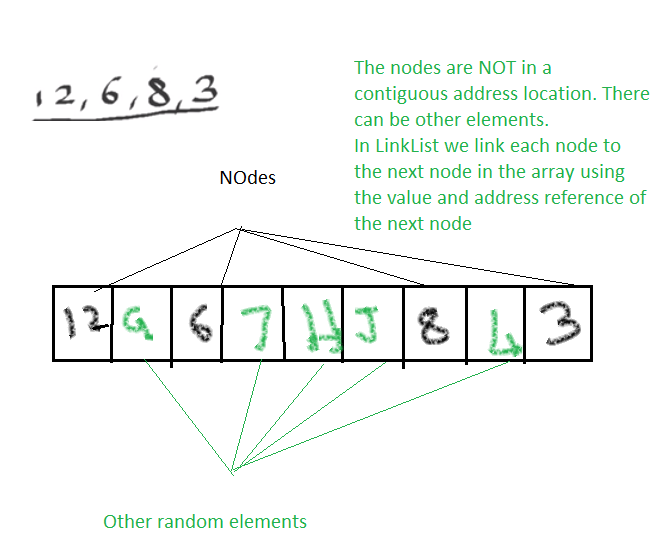
* Solution

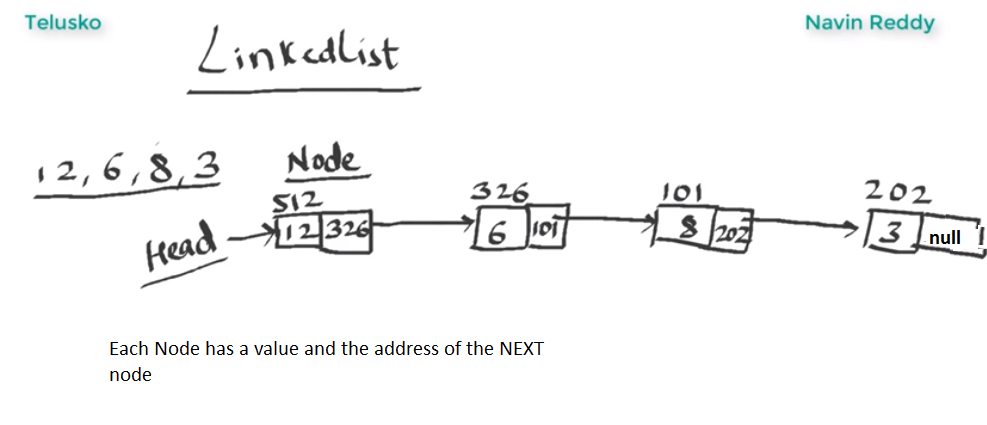
The above disadvantage is using a dynamic array called **LinkedList**



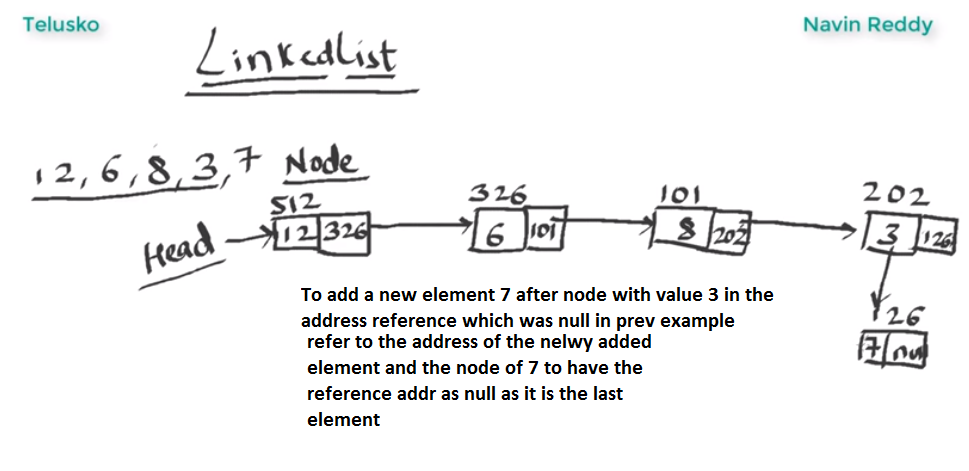
**Linked List**

* We can add/remove/update elements at runtime unlike the Arrays
* Every element is linked to each other
* The elements **need not** be in a **contiguous** location in memory.
* They are not in a sequence.
* Data and Reference concept.
* Each node has an info (value of the element) and a reference
* The first Node element is called the **Head**
* Each Node has an info and the address of the **NEXT** node.
* The last node’s reference **may** be null in a Single Linked list
* Single Linked List has only forward access to the node . No reverse access
* Double Linked List has both Forward and Backward /Reverse access to the node.
* Advantage
  + Insertion at any index is way easier than an Array
* Disadvantage
  + Fetching Slow compared to Array
  + Array it works on indexing as it is stored in a sequence (contiguous)
  + For searching it **used O(n)**

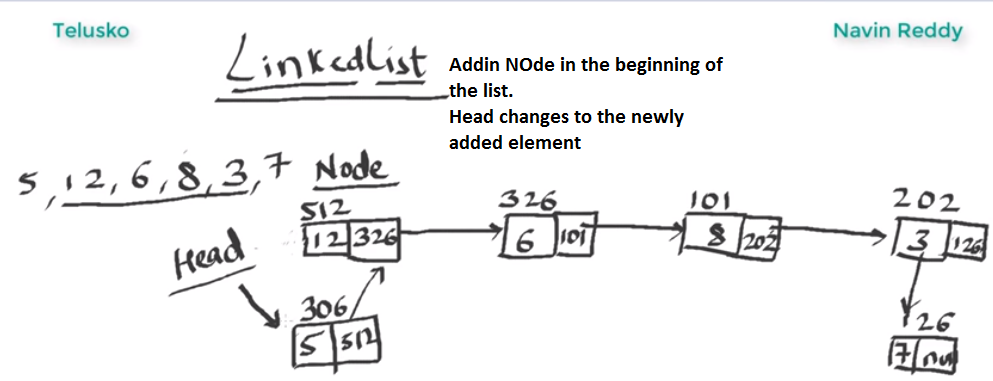


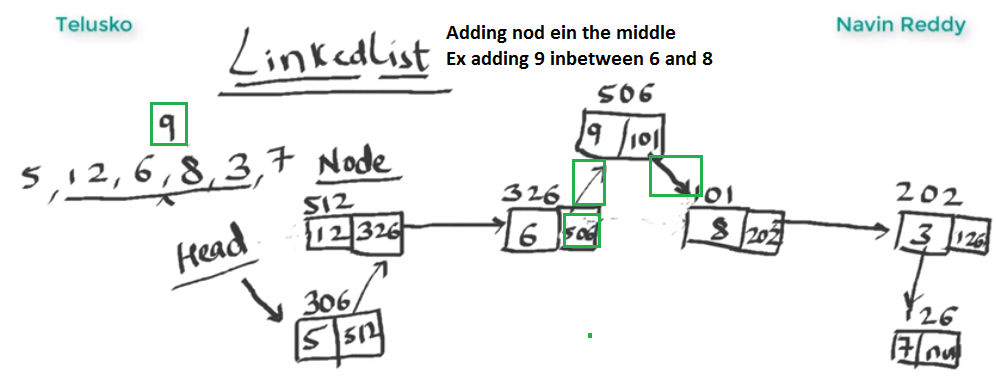


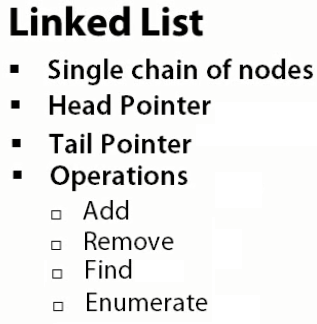
**Adding ELEMENT at the END**

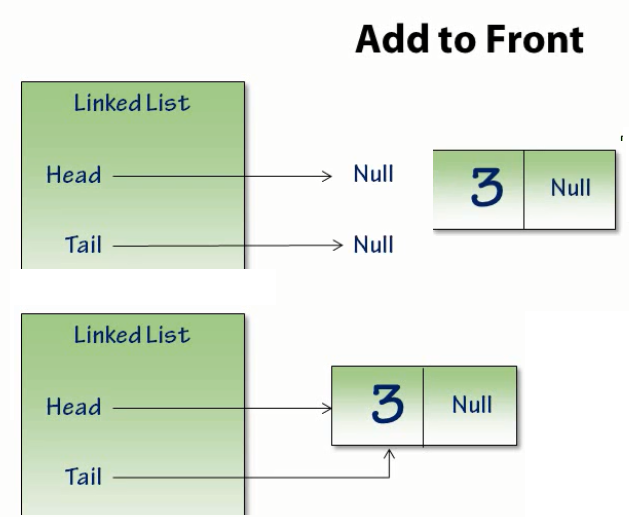


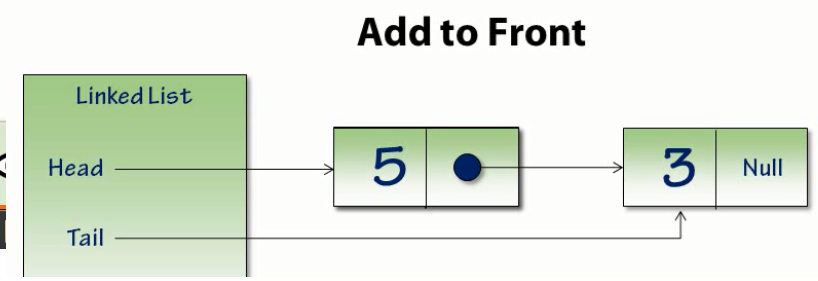
Adding element in the beginning of the node







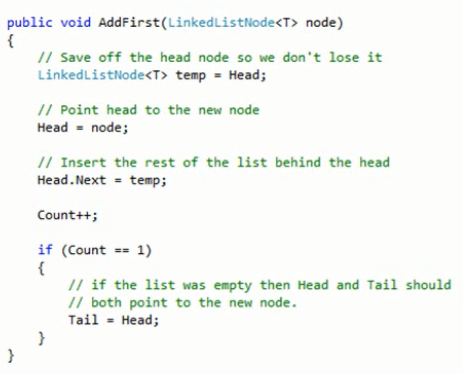




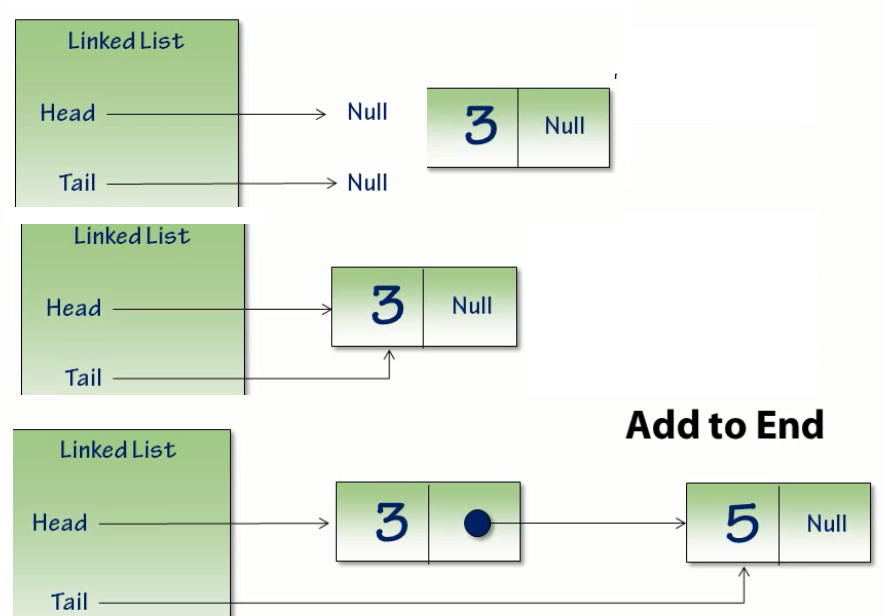
* All starts with an empty list.
* Linked List is a single node chain with Head and Tail node pointers, both of which have initial values of Null.
* The first step to adding the node is allocating the node.
* We've allocated the node with the value 3.
* Adding this node to the Linked List means first pointing the Head and Tail pointers at this node. Since the list has only one item, the Head and Tail pointers will both refer to that **same** item.
* We allocate a new node and point the Head node to it, pointing its Next pointer to the node that was previously the Head node.
* Since we're adding the node to the start of the list, we don't need to update the Tail pointer, it already points to the last node.

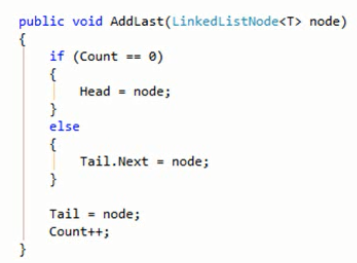
**Coding points**

* The very first thing we do is store the Head pointer into **a temporary variable**. It's the start of the existing node chain, and we'll want to keep our reference to that.
* The list's Head pointer is now updated to point at the node being **added** to the front of the list.
* The existing node chain, the temporary variable, is **unlinked** to the **new** Head node.
* We then do a little bookkeeping to track the **number of nodes** in the list.
* If there's only one node in the list, we set the **Tail node to be equal to the Head node**,
* Adding a new node to the front of the list only involved allocating the data and updating a few pointers.



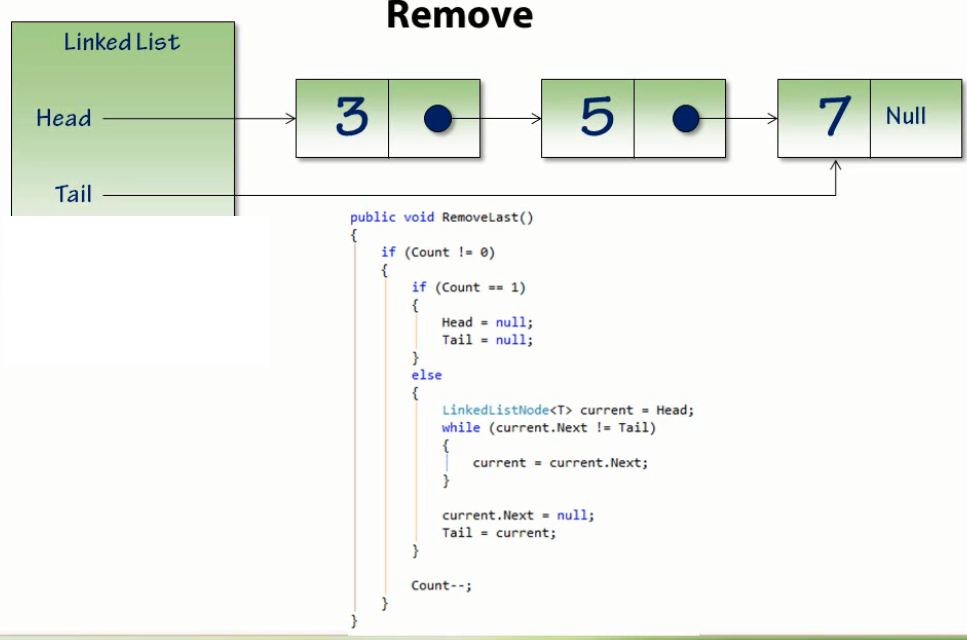
**ADD TO THE END OF THE NODE**

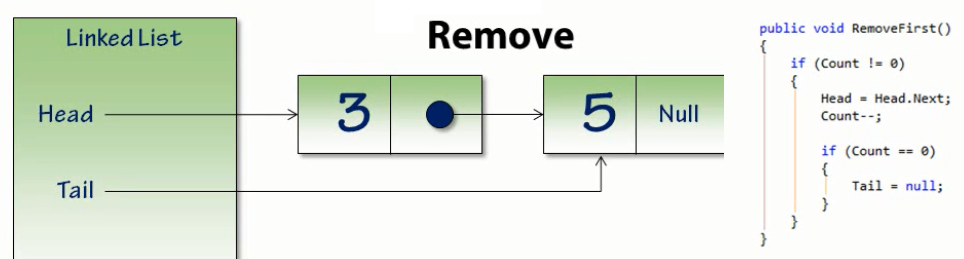


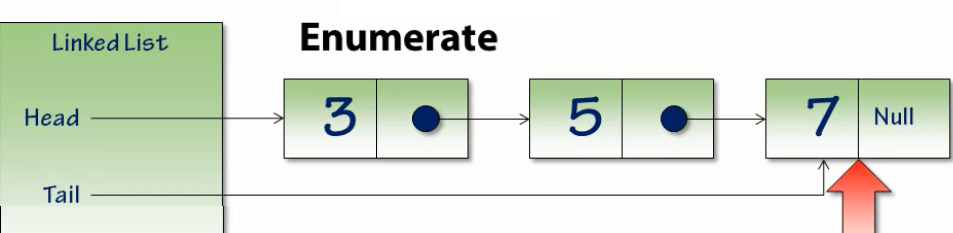


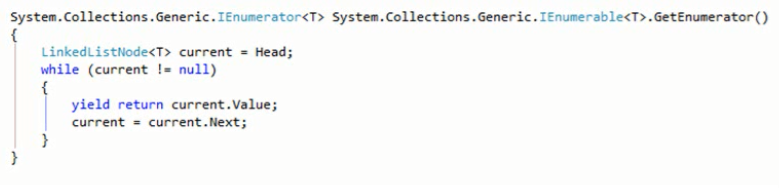
* We have an empty list with a Null Head and Tail pointers,
* It's that Tail pointer that's going to make this operation much easier.
* When we create our first node, the Head pointer is updated to point to the new node, and so is the Tail pointer.
* When a second node is created, the Head pointer remains unchanged.
* The Tail pointer points to the new node.
* Having the Tail pointer allows us to add the node to the tail very easily.
* There are only two cases we need to worry about, whether we're adding a node to an empty list or to an existing node chain.
* If the list is empty, we point the list Head pointer to the node being added.
* If the list is not empty, we chain the node being added to the end of the existing chain.
* In either case, the Tail pointer should now point to our added node
* We increment the counter, which is keeping track of how many nodes are in the list.

**REMOVE THE LAST NODE**

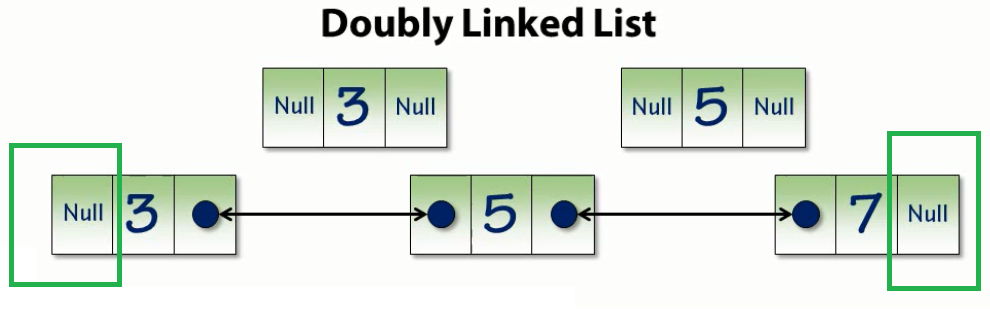
****

****

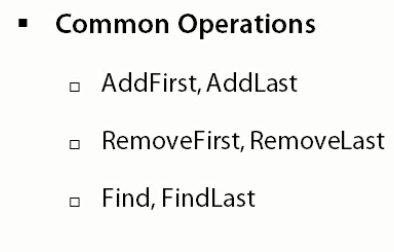
****



**Doubly Linked List**

****

* Doubly Linked List, each node contains two pointers; one to the **Next node** just like the Singly Linked List, and one to **the previous node.**
* A Doubly Linked List starts with the single node. A Single Node contains Null Previous and Null Next pointers.
* Just like with the Singly Linked List, we need to create a second node to start the node chain.
* Like the Singly Linked List, the Next pointer of the first node will now point to the second node.
* This is a Doubly Linked List, we'll also create a link back from the second node to the first.
* We can navigate from the second node back to the first node as easily as we can the first to the second.
* Node the Head node ‘s previous = null
* The last node’s next = null

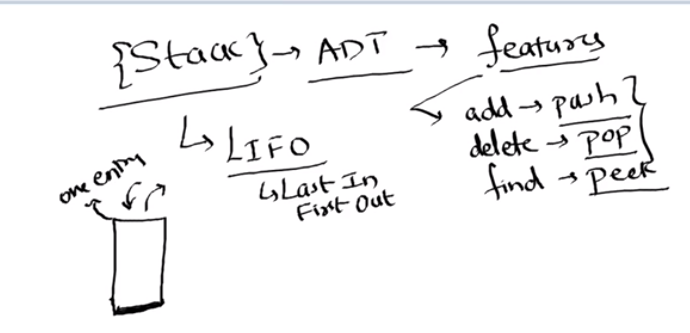


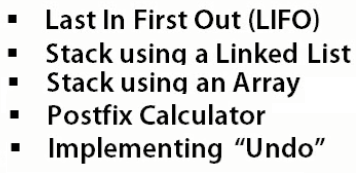
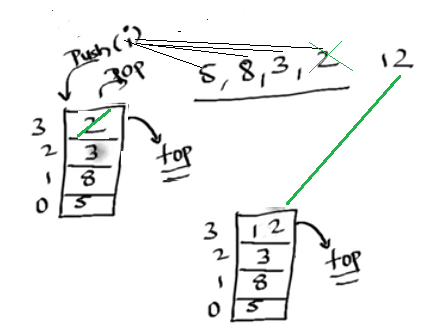
* addFirst
* addLast
* addItemAt
* removeFirst
* removeLast
* removeItem
* removeAtIndex
* enumerate



================================

STACK





* Stack is an A TD (Abstract Datatype we concentrate on the features it provides and not its implementation.
* Based on LIFO (the item inserted Last we can access it first) Last in first out. It has **only** one entry point. ie it is same entry and same exit point.

ex: Stack of books, we remove the last book is taken out to pick up the first book.

ex: In real world. When we do an alt tab we see a list of applications user has accessed.

Ex: Restaurant plate stacker where clean plates are stacked in a cylindrical steel holder.

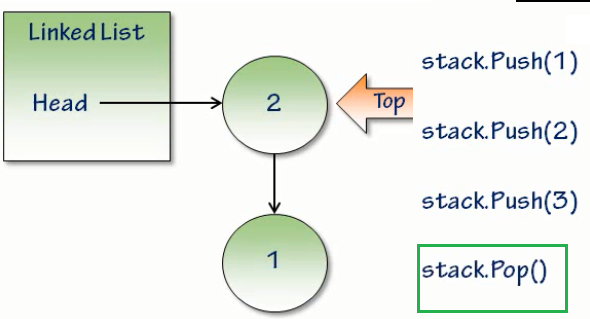
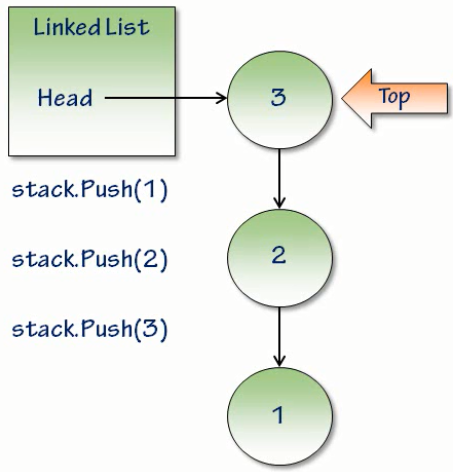
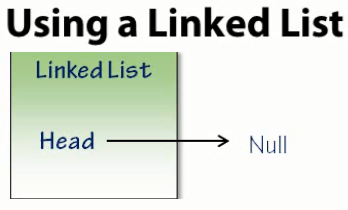
The application the user last accessed is seen first (ie last accessed first out/first seen)

* The pointer that references the top most element is called top.
* One can see the top most element in the stack which is called peek (peeking)
* Removing /Deleting the element from the stack is called a pop.
* Stack as mentioned is an ATD, so it can have different implementations using
  + Arrays : Fixed Array or Dynamic Array
  + Linked List

**Stack overflow exception**: In a Fixed array if we try to add an element more than the size of the array ie crossed the size of the array, we get stack overflow exception.

**Stack underflow exception**: If there are no elements and we try to pop(delete) an element we get stack underflow exception.

* Some methods
* add = push
* delete = pop
* find = peek
* isEmpty
* size



**Choosing LinkedList for stack implementation**

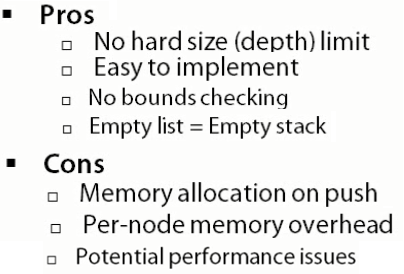
Why would I choose to use a Linked List as the data storage medium for a stack as opposed to another structure such as an Array?

**Advantages**

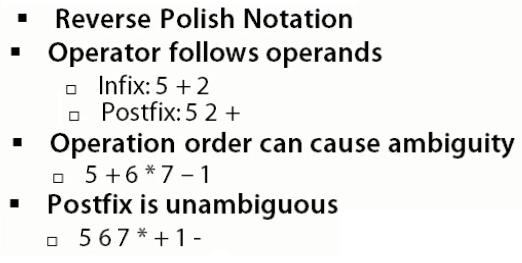
* Linked List, it has no hard size limit
* It's also a very straightforward way to implement a stack. An Array, for example, would require bounds checking to make sure that a push didn't cause the item to exceed the Array's bounds.

**Dis-Advantages using Link List**

* The first is that adding an item to the stack using Linked List causes a memory allocation **to occur every time** as compared to an array **where contiguous memory allocation takes** place during declaration. This can end up causing undesirable performance characteristics in high performance systems.
* Other issues include things **like data locality problems and memory fragmentation**; things that Linked Lists run into because they're storing the nodes **throughout the heap**, but an Array can avoid by keeping all the data **really near each other**
* Pre - node memory overhead : The **memory cost** for each node can be significantly more than the cost of the **data being stored**.
  + example, a 32-bit value such as an integer might have memory overhead several times larger than the integer itself.



POST FIX CALCULATER



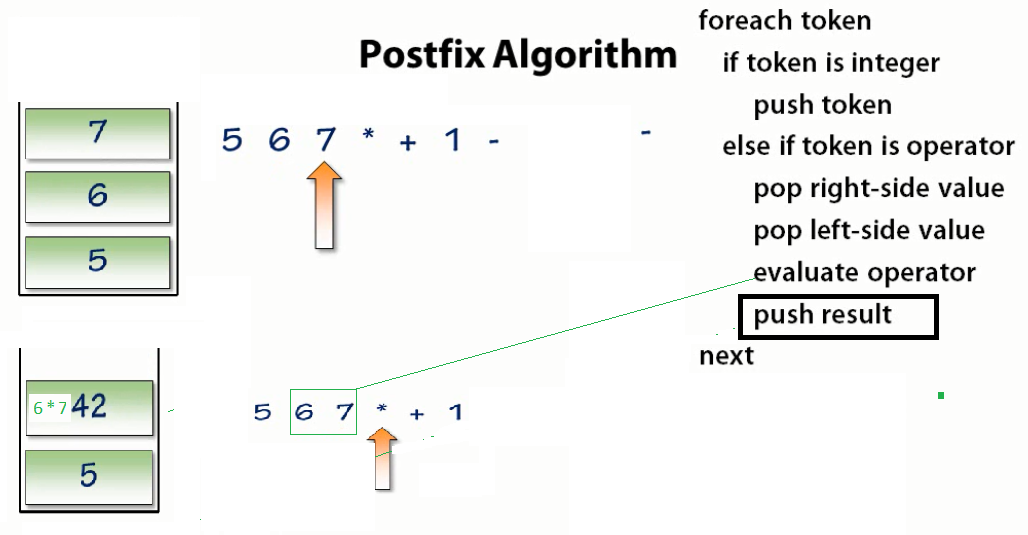
Algorithm for the postfix calculator:

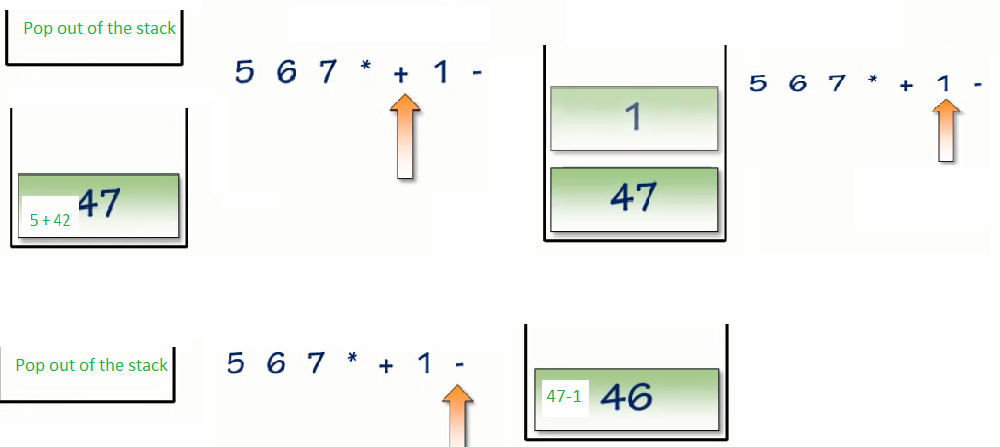
* The algorithm starts by **enumerating** over each token.
* We start with an empty Stack.
* If the token is an **integer**, the value is pushed
* If the token is an **operator**, the start popping elements from the stack
* The popped element goes **to RHS of the operator** and the other element on the **LHS of the operator**
* The operation is performed and the result is then **pushed** into the stack

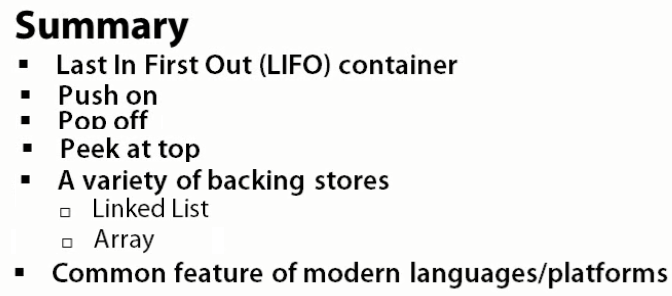
In our example



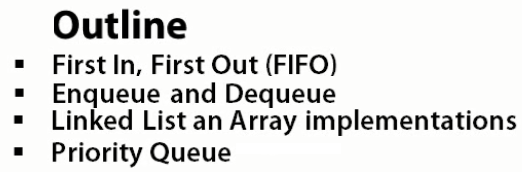
* 5 is an integer, so we'll push it to the Stack.
* 6 is an integer, so we'll push it to the Stack.
* Next token 7 is an integer, it's also pushed onto the Stack.
* The next token **\*** , is NOT an integer, it's an operator:
* Start **popping** things off the stack.
* We're going to pop a 7 off, and that will become the **right-hand side of the multiplication expression**
* And then we'll pop the 6 off, which will be the left-hand side; 6 x 7 will multiply together to give the value 42, which we'll then **push onto the Stack.**
* Then we continue our process.
* We go onto the next token, which is the **+** operator,
* **Pop** the right value, the 42 from the Stack
* Pop the 5 from the stack. We'll **add** them together, give it a value of 47, which we **push** back onto the Stack.
* The next token is an integer, we **push** that onto the Stack, and then the next token is an operator **subtraction**.
* So, we'll pop the 1, this will be the right-hand side of the subtraction expression, and then we'll pop the 47. Forty-seven - 1 is 46, which we push onto the Stack, and that's our answer. Now that's our answer because we've gotten to the end of the token list, and the last item on the Stack is the result.







QUEUE

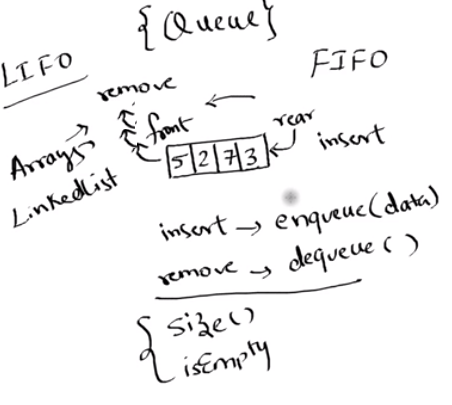


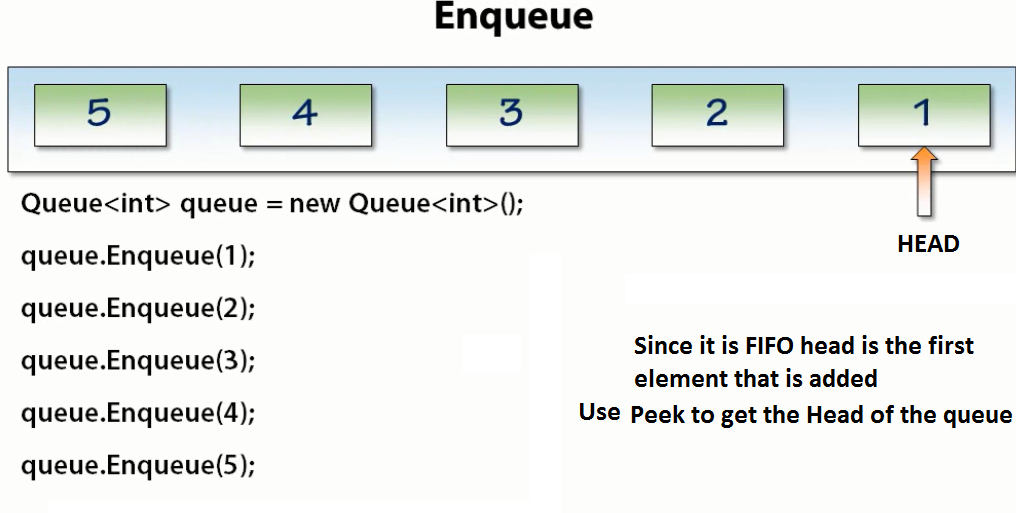
Queue :

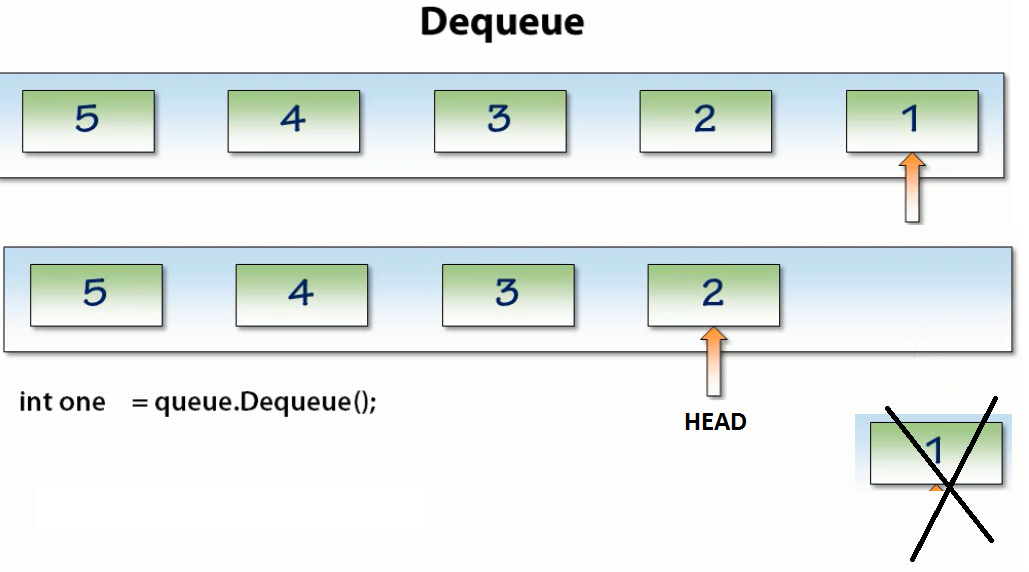
* FIFO (First in First out) collection. A Queue is a collection that returns items in the **same order that they were added**.

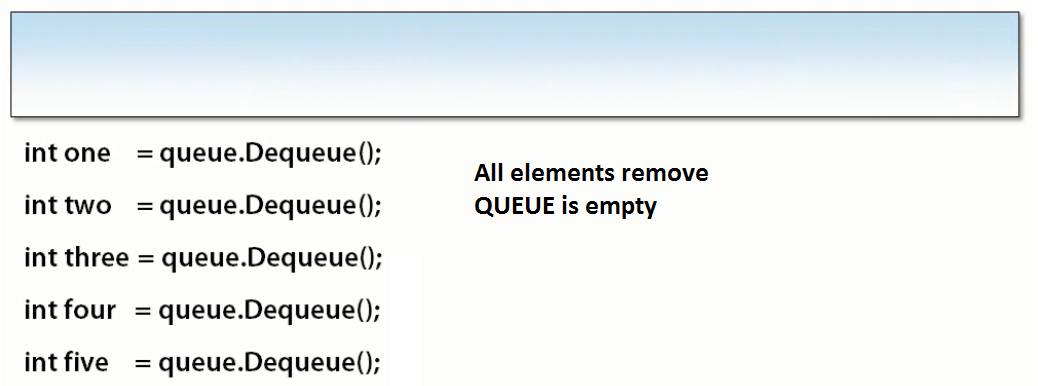
Ex: Checkout line at a grocery store. The people in the line are checked out in the order that they get into the line; first in the line, first checked out, last in the line, last checked out.

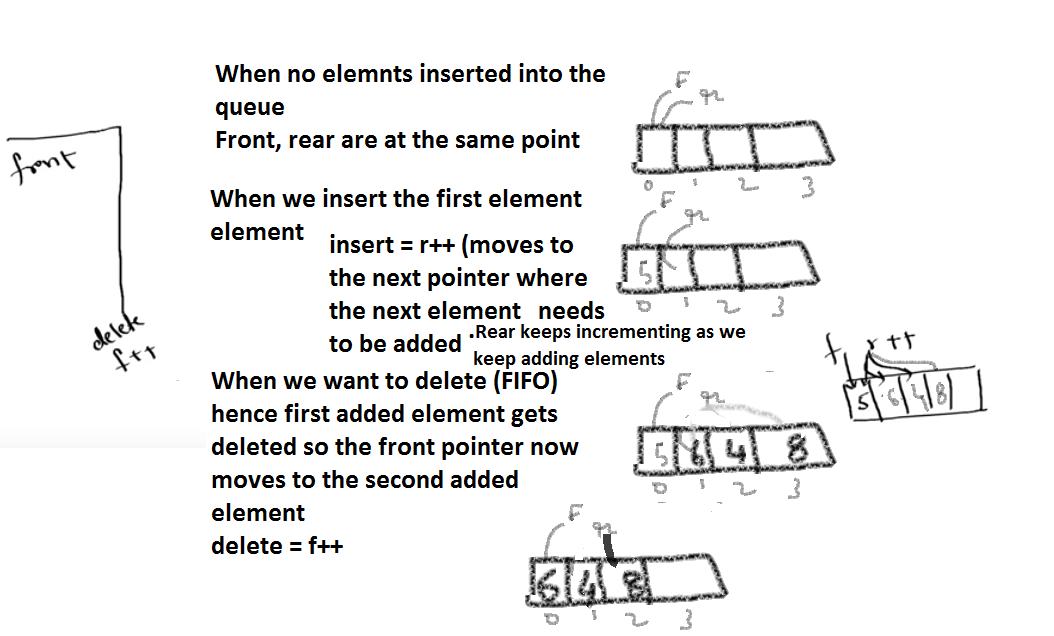
* Enqueue is to add elements into the queue
* Dequeue is to remove/delete elements from a queue
* The first element added into the queue is called the head of the queue. As it follows FIFO











**QUEUE USING CIRCULAR ARRAY**

