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)

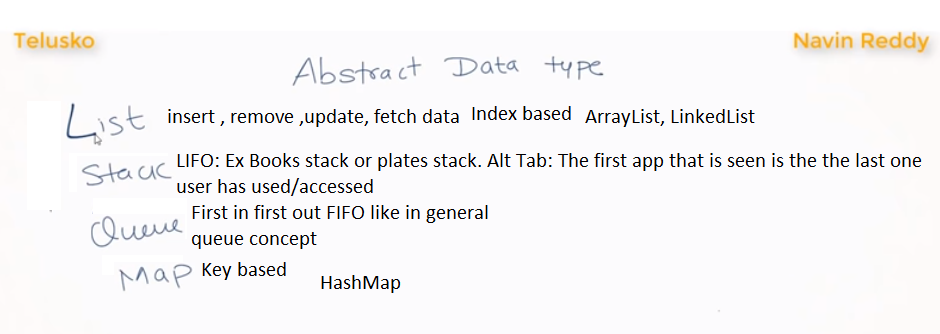
**Why Data Structures?**

* 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**

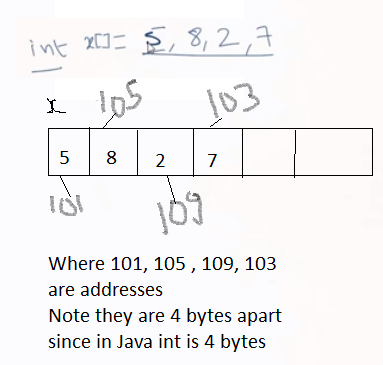
We cannot change the size of the array once the array is declared.

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

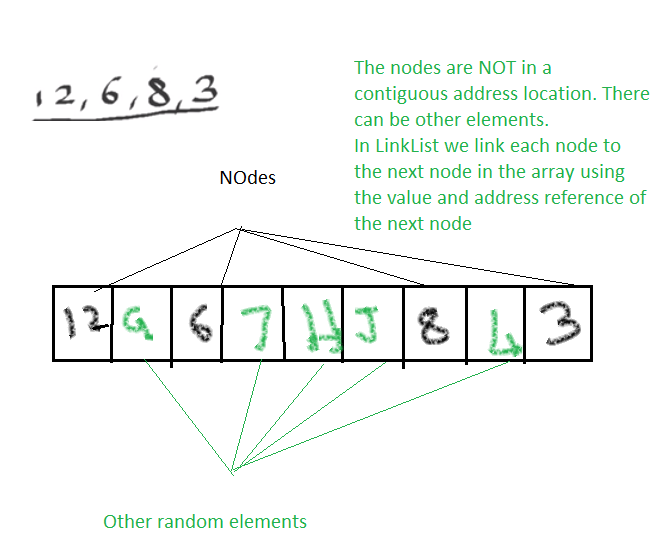
* Solution

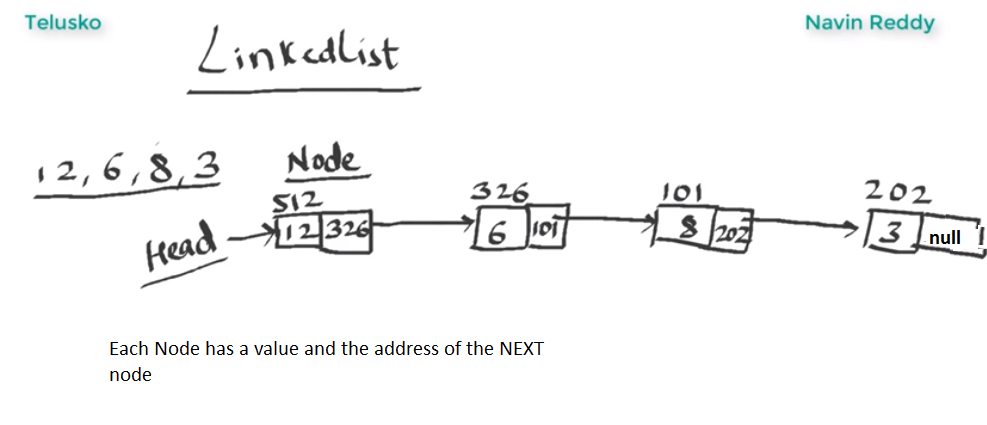
The above disadvantage is using a dynamic array called **LinkedList**



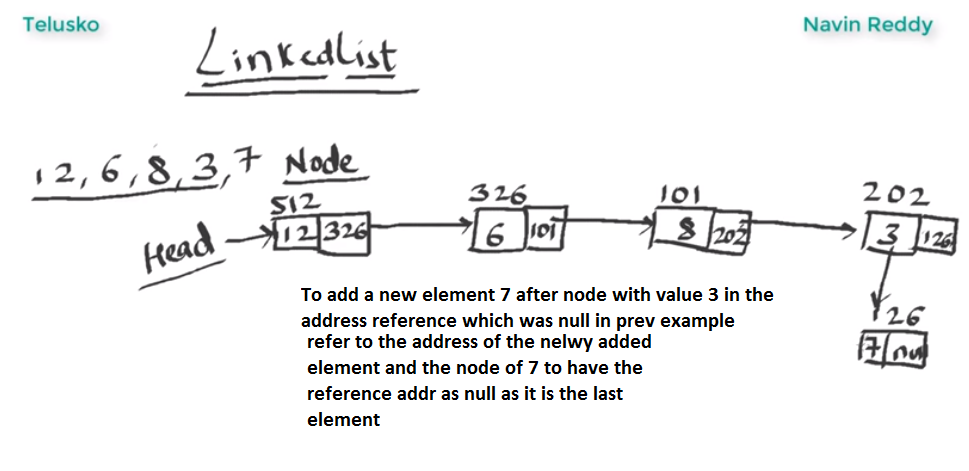
**Linked List**

* 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)**



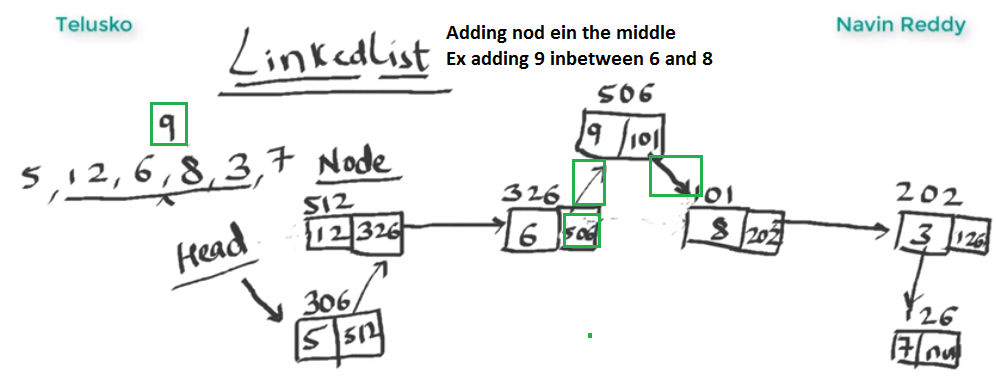


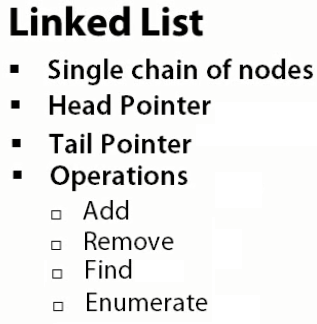
**Adding ELEMENT at the END**

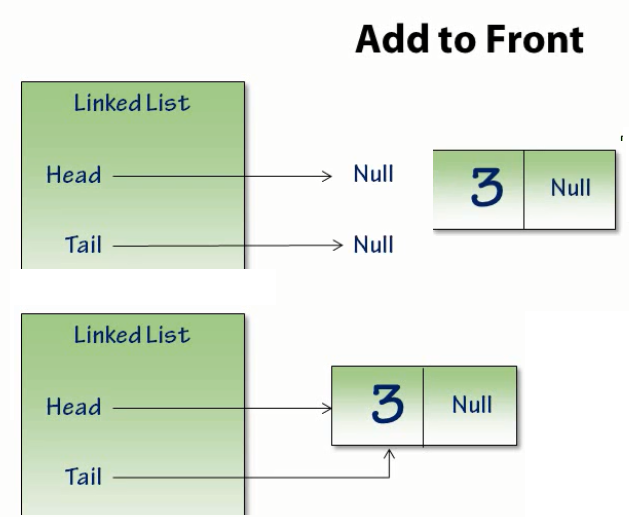


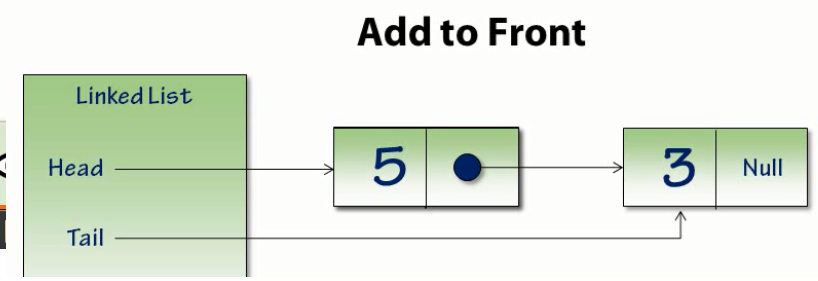
Adding element in the beginning of the node







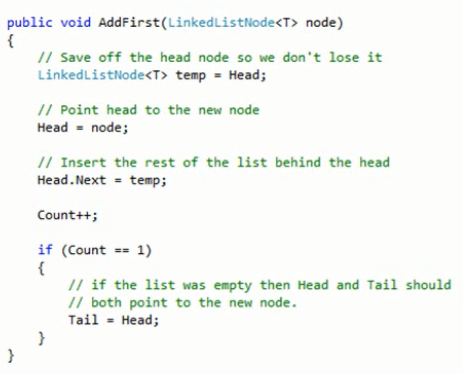




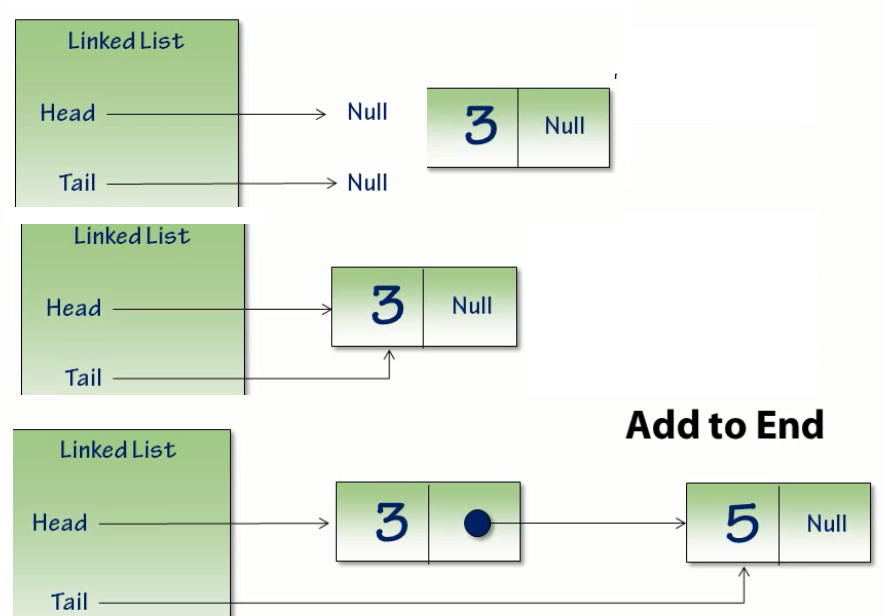
* All starts with an empty list.
* Linked List is a single node chain with Head and Tail node pointers, both of which have initial values of Null.
* The first step to adding the node is allocating the node.
* We've allocated the node with the value 3.
* Adding this node to the Linked List means first 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.

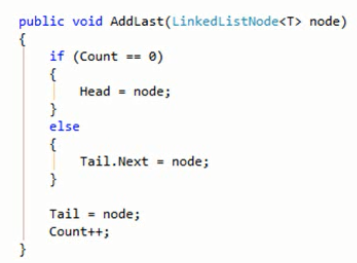
**Coding points**

* 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.



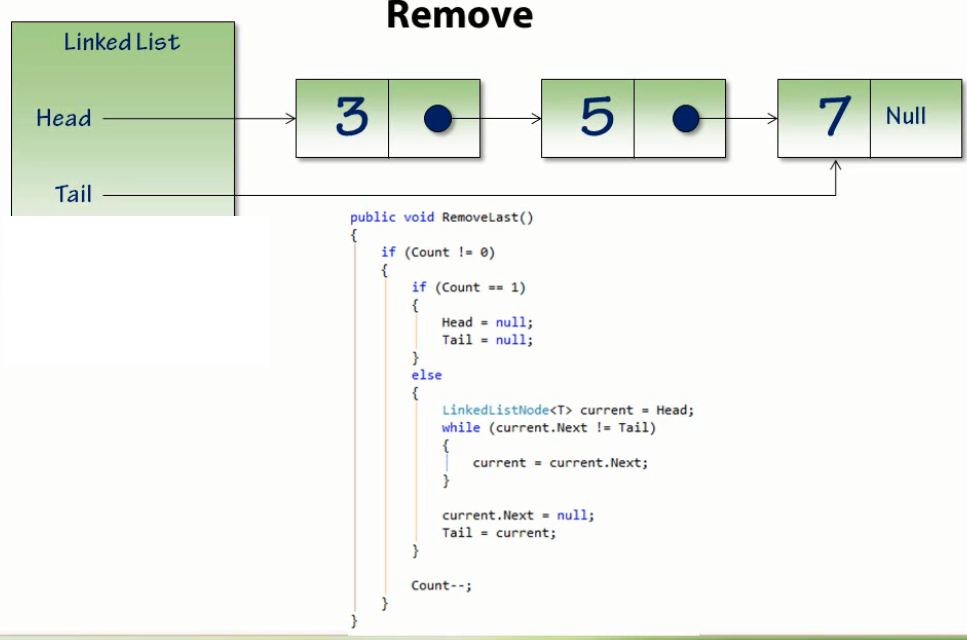
**ADD TO THE END OF THE NODE**

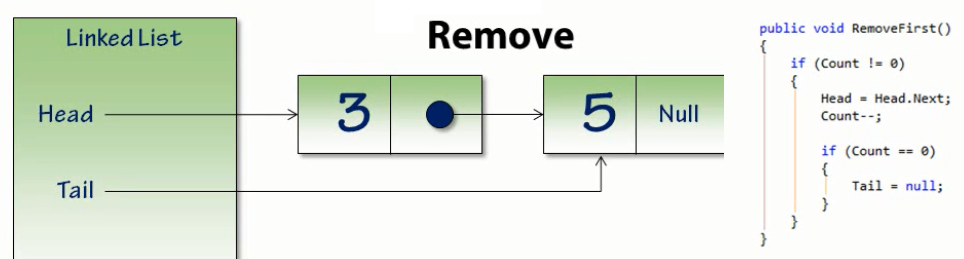


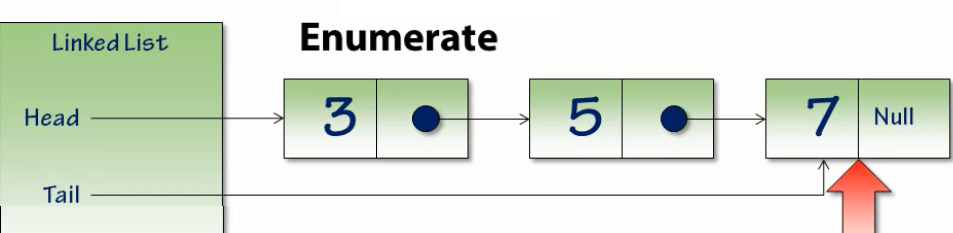


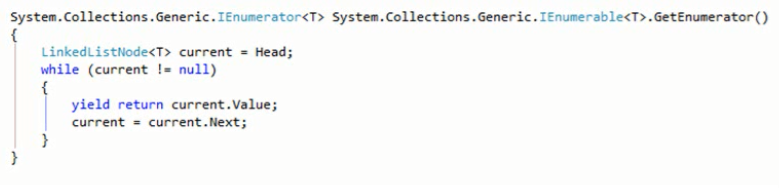
* We have an empty list with a Null Head and Tail pointers,
* It's that Tail pointer that's going to make this operation much easier.
* When we create our first node, the Head pointer is updated to point to the new node, and so is the Tail pointer.
* When a second node is created, the Head pointer remains 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**

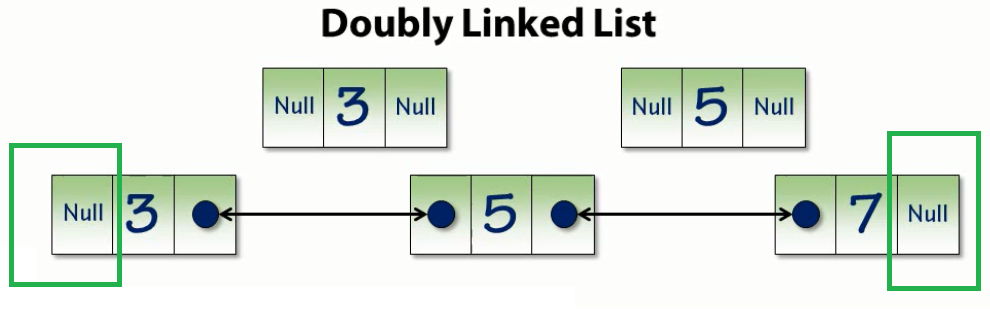
****

****

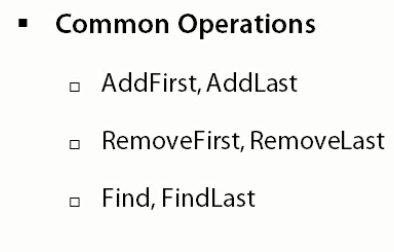
****



**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

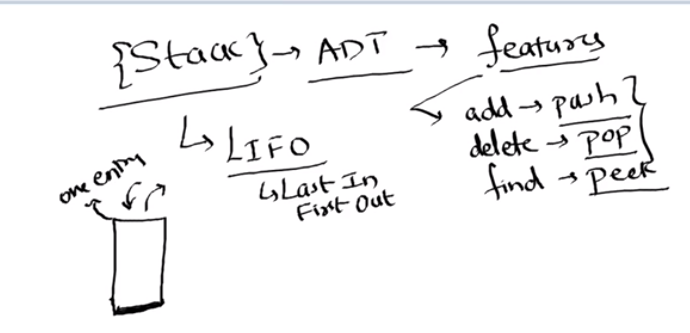


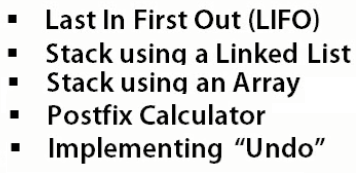
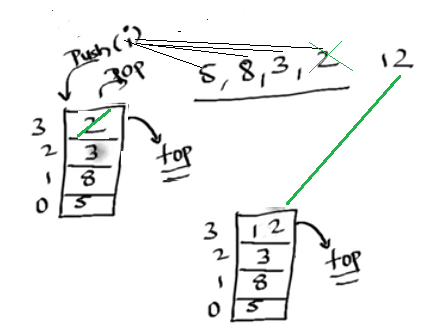
* addFirst
* addLast
* addItemAt
* removeFirst
* removeLast
* removeItem
* removeAtIndex
* enumerate



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

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.

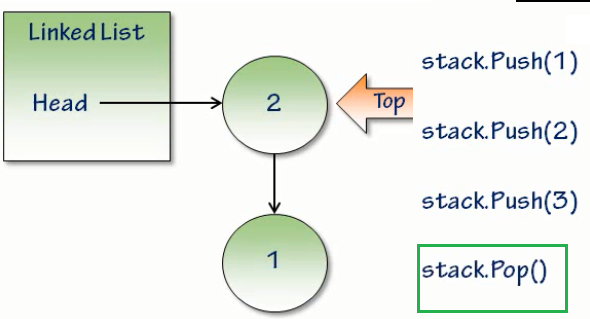
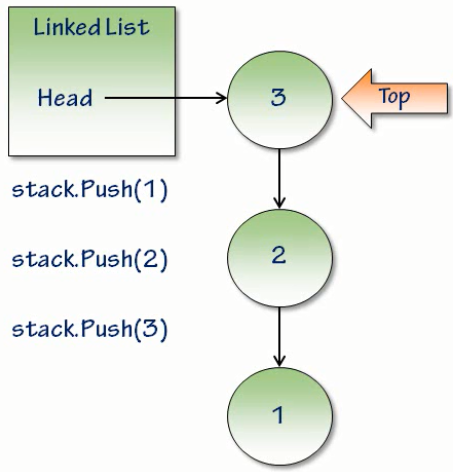
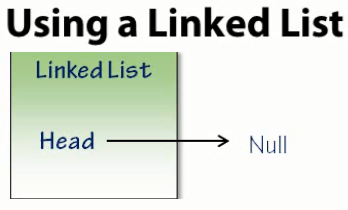
The application the user last accessed is seen first (ie last accessed first out/first seen)

* 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**

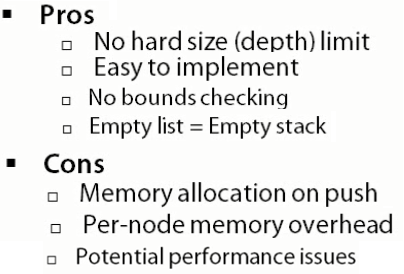
Why would I choose to use a Linked List as the data storage medium for a stack as opposed to another structure such as an Array?

**Advantages**

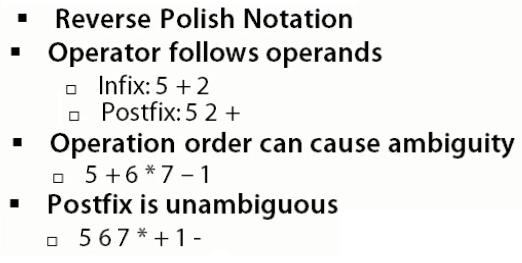
* Linked List, it has no hard size limit
* It's also a very straightforward way to implement a stack. An Array, for example, would require bounds checking to make sure that a push didn't cause the item to exceed the Array's bounds.

**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



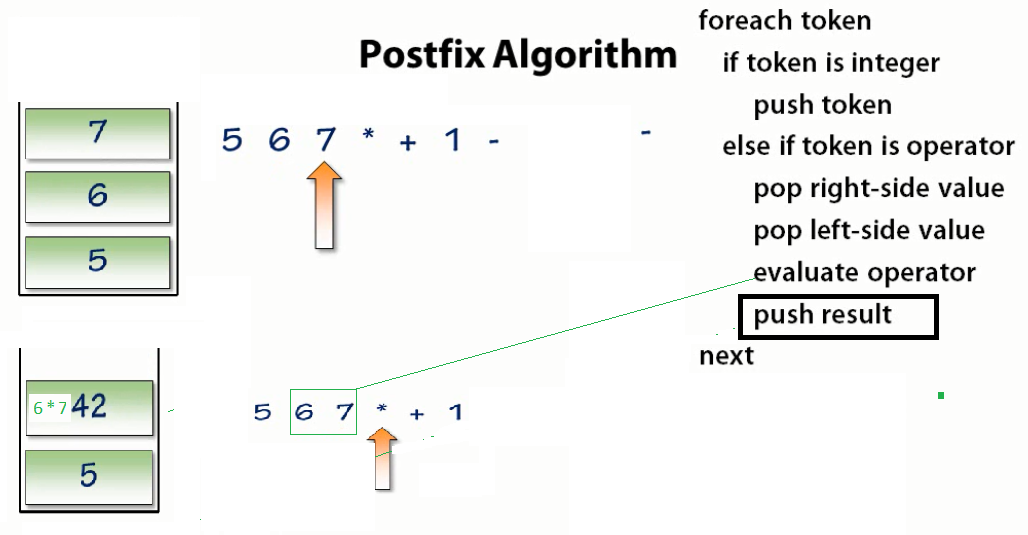
Algorithm for the postfix calculator:

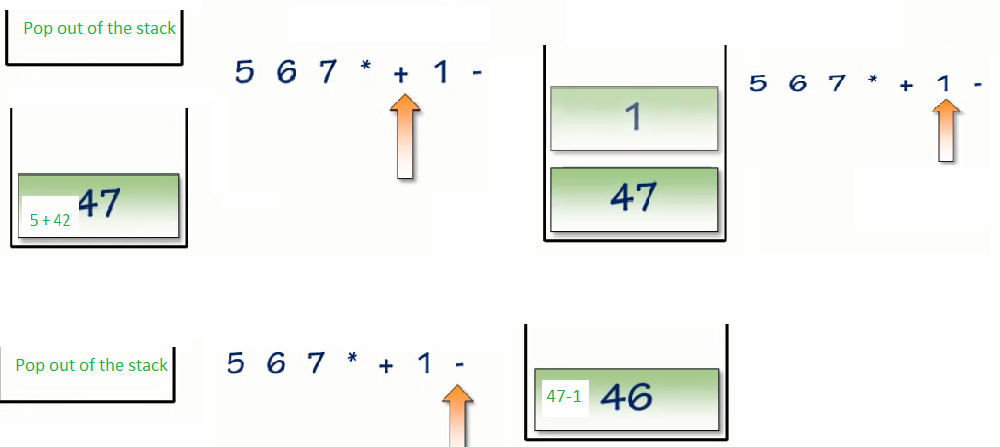
* The algorithm starts by **enumerating** over each token.
* We start with an empty Stack.
* If the token is an **integer**, the value is pushed
* If the token is an **operator**, the start popping elements from the stack
* The popped element goes **to RHS of the operator** and the other element on the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

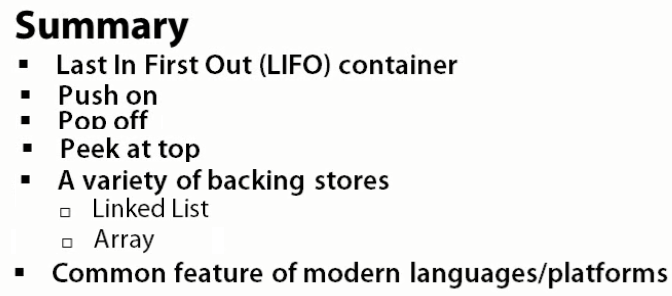
In our example



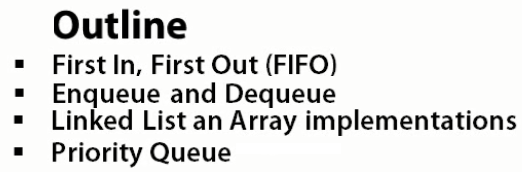
* 5 is an integer, so we'll push it to the Stack.
* 6 is an integer, so we'll push it to the Stack.
* Next token 7 is an integer, it's also pushed onto the Stack.
* The next token **\*** , is NOT an integer, it's an operator:
* Start **popping** things off the stack.
* We're going to pop a 7 off, and 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

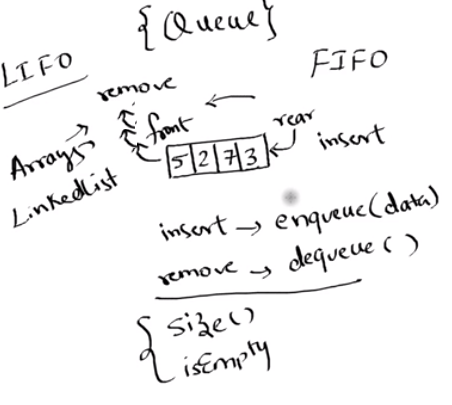


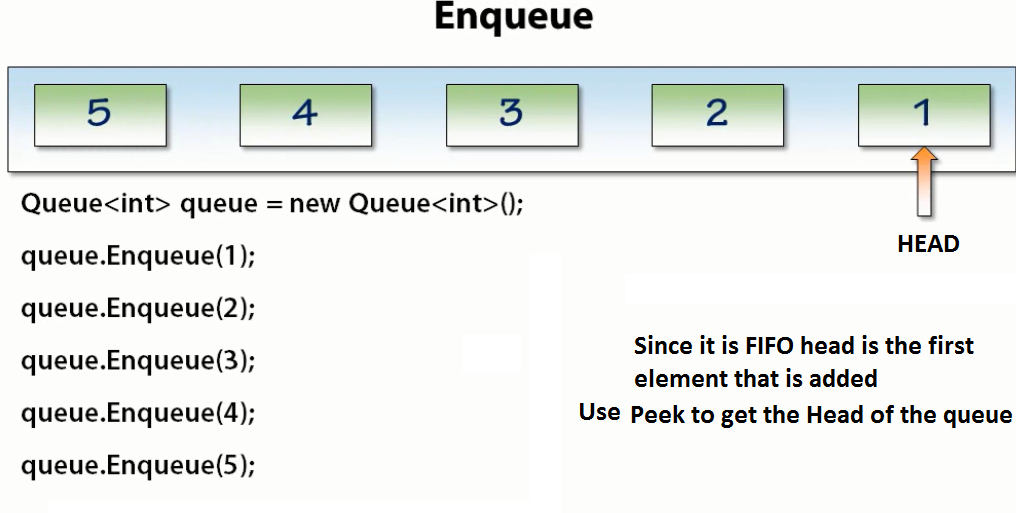
Queue :

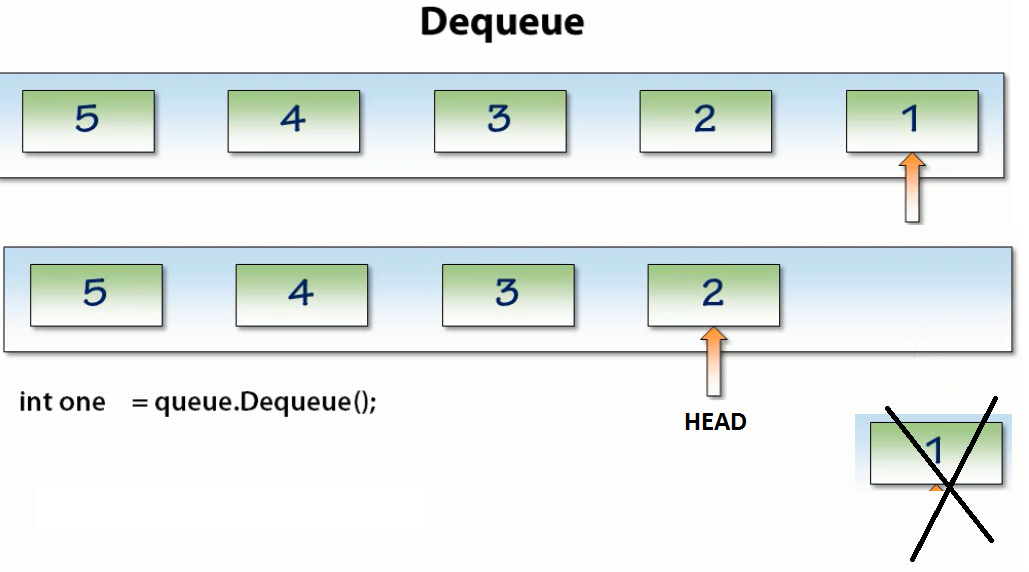
* FIFO (First in First out) collection. A Queue is a collection that returns items in the **same order that they were added**.

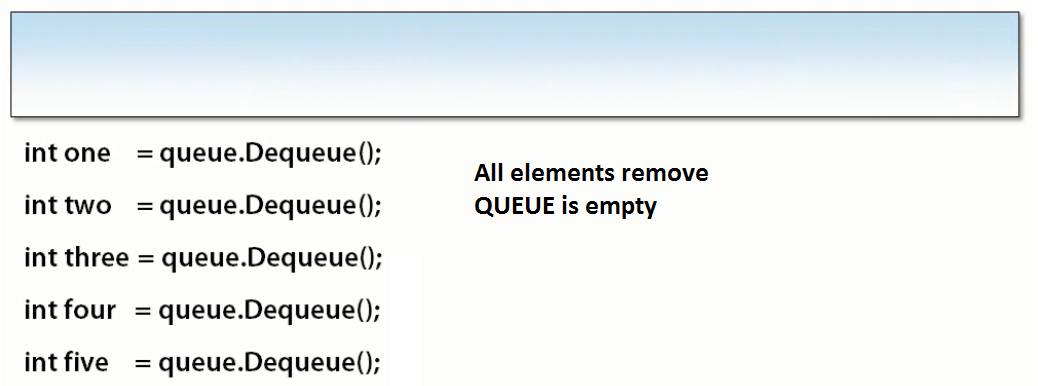
Ex: Checkout line at a grocery store. The people in the line are checked out in the order that they get into the line; first in the line, first checked out, last in the line, last checked out.

* 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



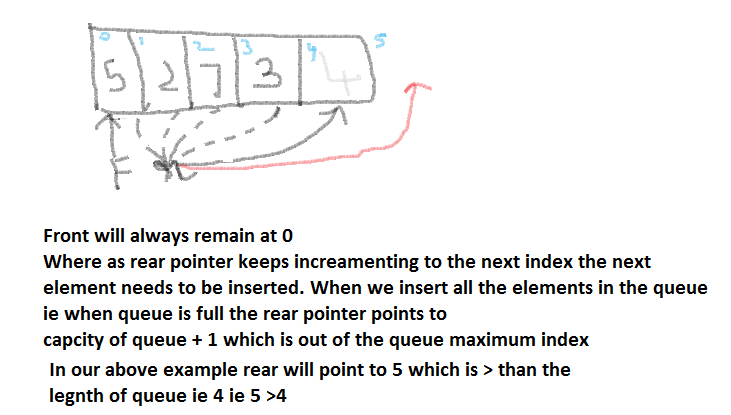


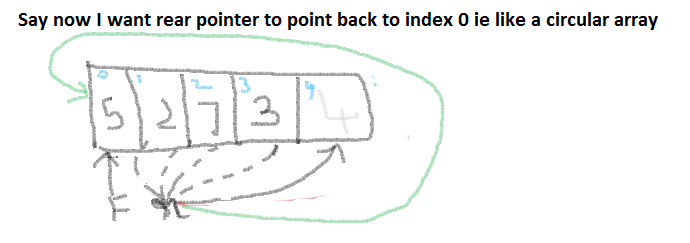


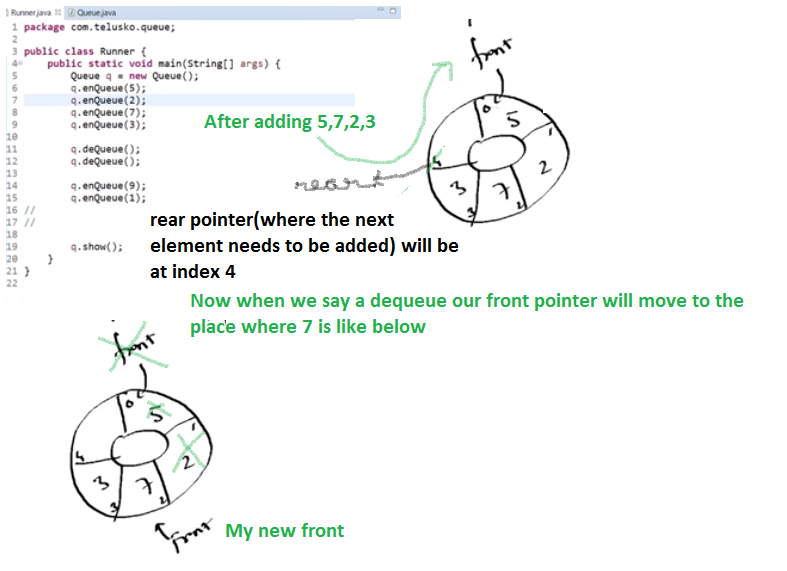


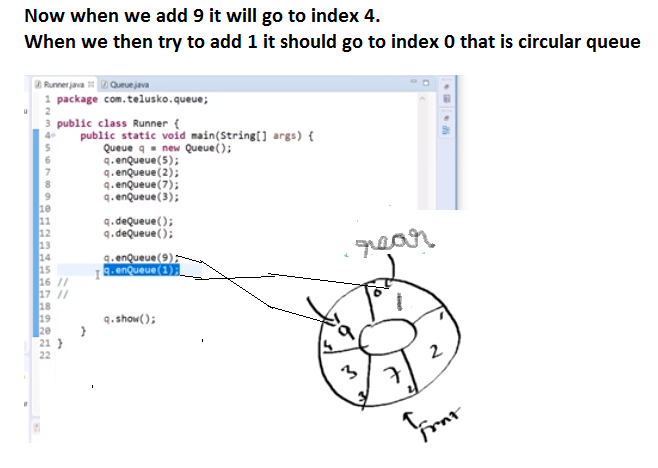


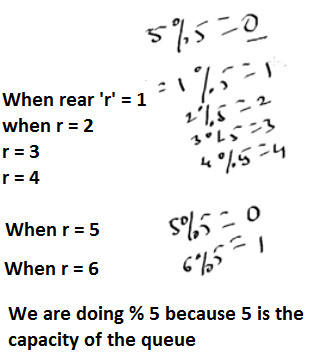
**QUEUE USING CIRCULAR ARRAY**

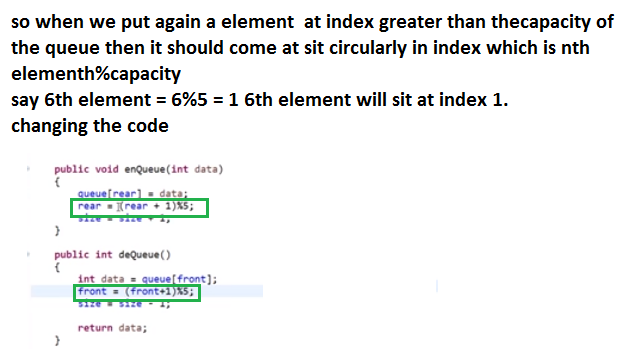












**PRIORITY QUEUE**

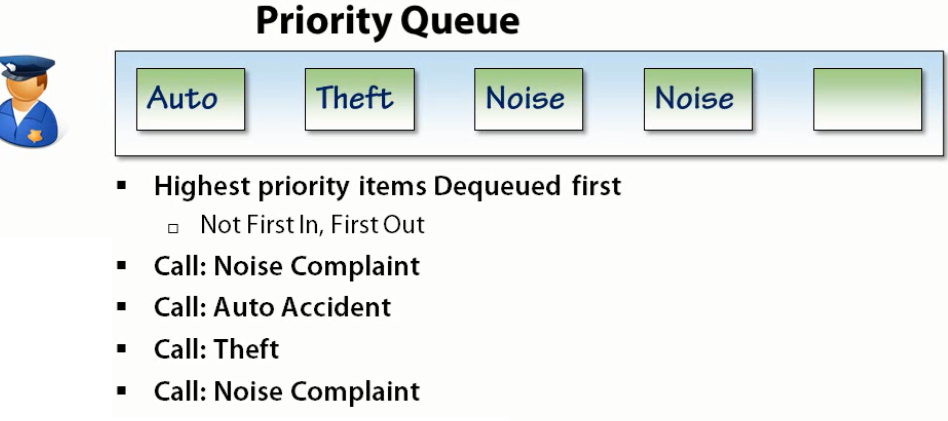
* Priority Queue is a specialization of the Queue, where the item is stored in priority order.
* Priority Queues differ from normal Queues in that they are not First In, First Out.
* They return the highest priority items first, regardless of the order in which they were **added** to the Queue.
* The first item is the one with the highest priority.

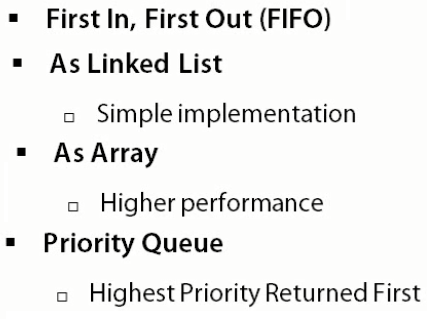
Example

* A police station call center.
* People call the police for all sorts of reasons, and each reason has a specific priority.
* Life and death issues are a higher priority than complaints about noisy neighbors.
* Officer starts his shift, he has nothing in his Queue.
* A call comes in with a noise complaint. Since it's the first call of the day, it's immediately the highest priority item in the officer's Queue.
* Then an auto accident, and this now becomes his focus and the noise complaint falls further in the Stack.
* A call comes in from a store that has experienced a theft. This is more important than a noise complaint, but not as important as the auto accident.
* Finally, another noise complaint comes in, and being of equal priority to the existing noise complaint, it's added to the Queue at the same location.
* So, you can see how a Priority Queue ensures that eventually every issue will be addressed, but the issues will be addressed in priority order.

Code Algorithm

* Add an item to the queue in priority order
* If the list is empty then , just add it
* else
  + Find the proper insert point.
  + If the current value is null then we have reached end of the list
  + If the current one being added is less than or equal ot the one being added
  + then add it at the behind
* Note Highest Priority queue is Dequed/Removed first , but they do not follow FIFO
* At any point the Queue should be in descending order where the Head is of highest priority



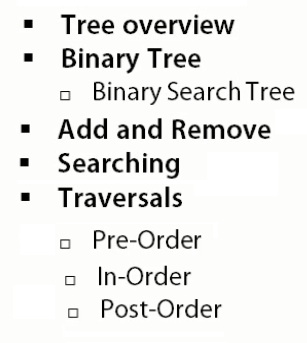


**TREE**

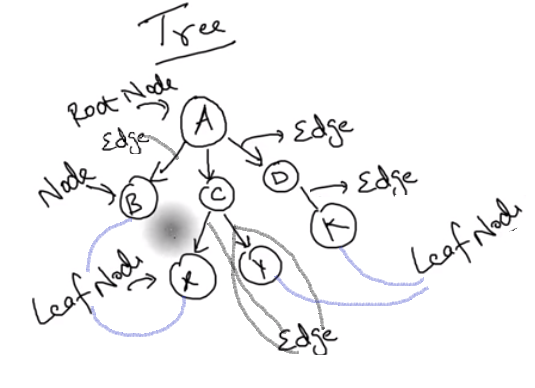
* Tree is an ADT.
* Tree has nodes
* Tree: Chained together nodes in a **Hierarchical order** unlike a linear manner like the linked lists.
* The point /or the node where the **traversal starts** is called the **ROOT node**.
* Root Node is the topmost of the Tree. There is only one Root node.
* Root node is called top node or the Head node.
* Root node has other child nodes.
* The nodes are connected to each other via a **link called as the Edge**.
* There is **exactly one path from the root node** **to any other node** in the Tree, and likewise exactly one path from any node in the Tree back to the root node, and therefore, there is exactly one path that can be taken between any nodes in the Tree. That limitation, a single path between nodes, is a fundamental rule that the Tree structures adhere to.
* If a node **does not have any child nodes** then that node becomes a **leaf node** or the **terminal** nodes
* **Depth** of a node is the number of **nodes above it in the tree** from which it descended.
* **Height** of the node is the maximum number of children it has say in its left or right traversal.
* The depth of the root node which is 0

Example of a tree structure can be seen in an organization like below:





**BINARY TREE**

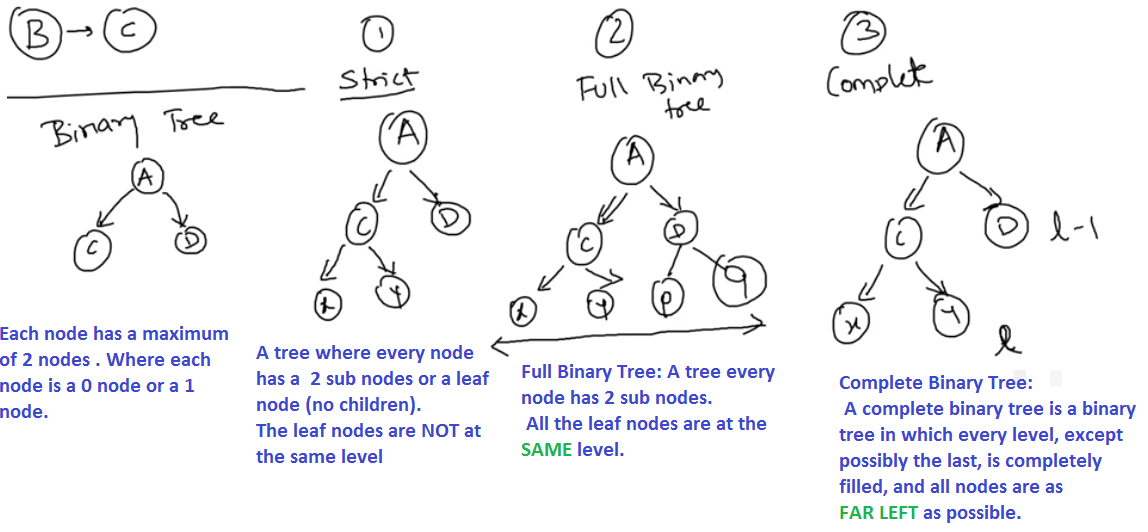


**Binary Tree:**

* Each node has a maximum of 2 nodes.
* Where each node is a 0 node or a 1 node.
* A Binary Tree is a Hierarchy of Data with some structure rules.
* It starts out with a Root Node. This is a node that has no parent
* We can create zero, one, or two children.
* Each child is itself a tree with the exact same structure limits(0 to 2 nodes) as the parent.
* Binary Tree has at most two child nodes, thus the name Binary, and those children are known as the Left and the Right Children.

**Types of Binary Trees**

* **Strict Binary:** A tree where every node has 2 sub nodes or a leaf node (no children). The leaf nodes are NOT at the **SAME LEVEL**
* **Full Binary Tree**: A tree every node has 2 sub nodes. All the leaf nodes are at the **SAME LEVEL**
* **Complete Binary Tree:** A complete binary tree is a binary tree in which every level, **except possibly the last**, is completely filled, and all nodes are as **FAR LEFT** as possible.

****

**BINARY SEARCH TREE**

A Binary Search Tree doesn't change the structural rules of the Binary Tree, but it imposes an additional data rule, and that is that all the values in the Tree are stored in Sort Order.

* The **smallest** values are on the **left**.
* The **largest** values are on the **right**.

Example

We start with the Root Node. In this case, the node has the value 4.

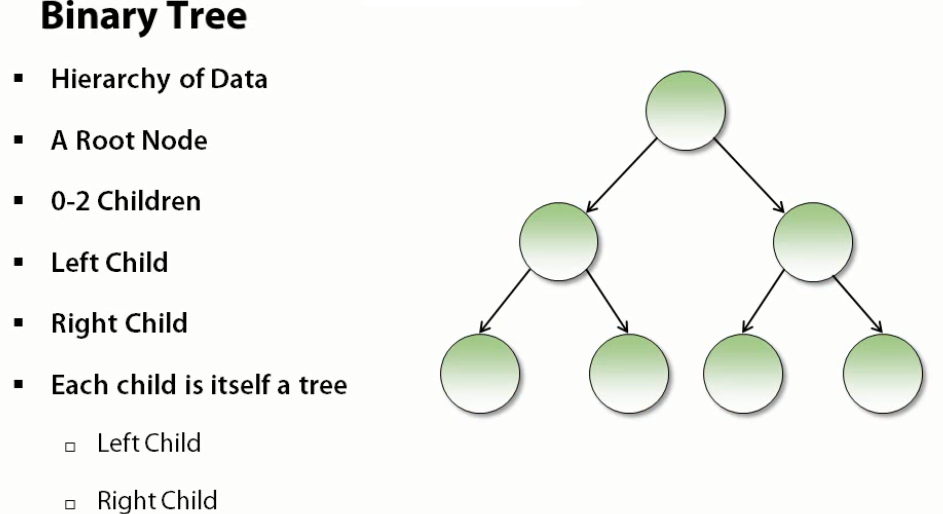
The Root Node has Child nodes with values 2 and 6

Value 2 is less than 4, it becomes the Left Child of the 4 node.

Node with value 6 is greater than 4, it becomes the Right Child of the Root Node. This rules is followed recursively throughout the Tree.

We can see that it's sorted in a way that the left-most node contains the smallest value, and the right-most node contains the largest. This is an invariant rule in the Binary Search Tree, and one that we'll rely on going forward in the module.

****

****

* A Binary Tree is a Hierarchy of Data with some structure rules.
* It starts out with a Root Node. This is a node that has no parent
* We can create zero, one, or two children.
* Each child is itself a tree with the exact same structure limits(0 to 2 nodes) as the parent.
* Binary Tree has at most two child nodes, thus the name Binary, and those children are known as the Left and the Right Children.
* A Binary Search Tree doesn't change the structural rules of the Binary Tree, but it imposes an additional data rule, and that is that all the values in the Tree are stored in Sort Order.
* The smallest values are on the left.
* The largest values are on the right.

Example

We start with the Root Node. In this case, the node has the value 4.

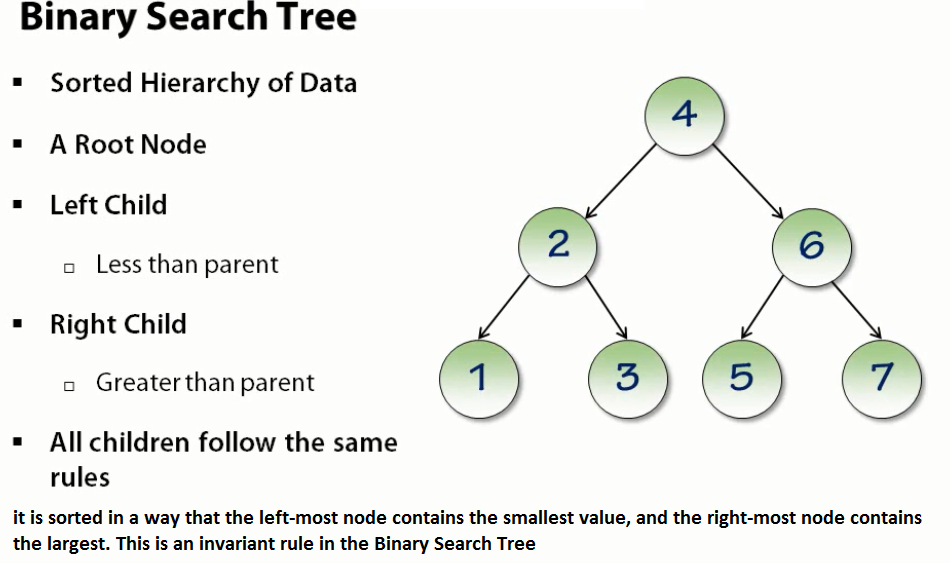
The Root Node has Child nodes with values 2 and 6

Because value 2 is less than 4, it becomes the Left Child of the 4 node.

Node with value 6 is greater than 4, it becomes the Right Child of the Root Node.

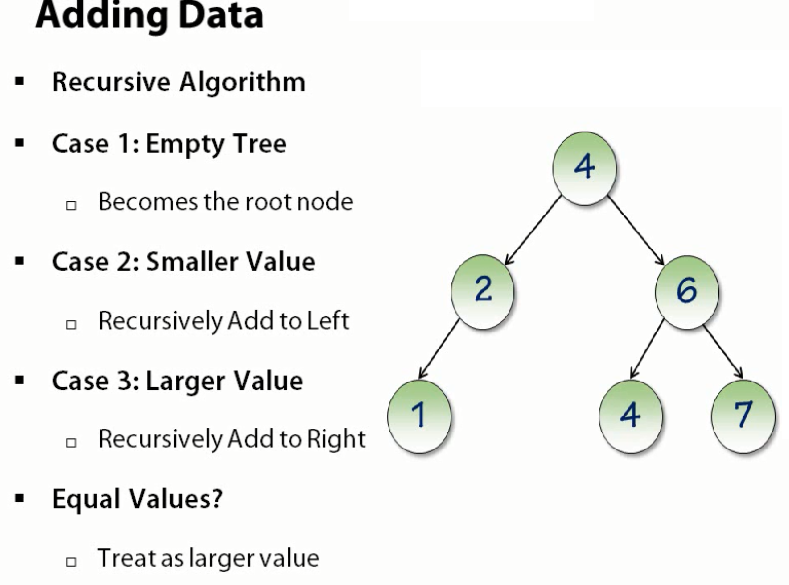
This rules is followed recursively throughout the Tree.

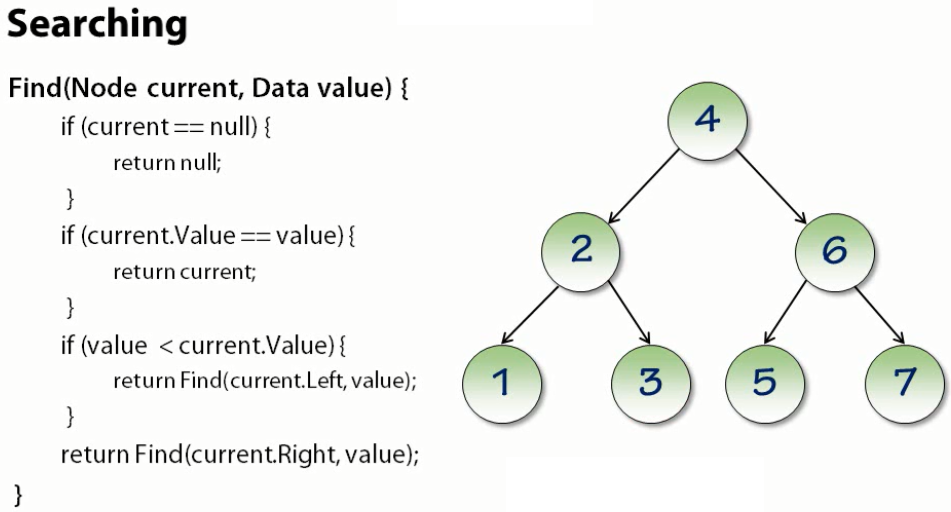
We can see that it's sorted in a way that the left-most node contains the smallest value, and the right-most node contains the largest. This is an invariant rule in the Binary Search Tree, and one that we'll rely on going forward in the module.



**Adding Nodes:**

* Adding items to the Tree is performed with a **recursive algorithm**.
* Initially we have an empty tree, which becomes the Root Node.
* In this case we've added the value 4.
* Now we want value 2; smaller than 4. Because it is a smaller value, the value is added to the left of the Root Node.
* If we add something even smaller it will be **recursively added to the left**. Since 1 was smaller than 4, it had to go to the left, and since 1 was smaller than 2, it had to go to that left of 2.
* If we add an even larger value, it will be recursively added to the right.
* 7 is larger than 4, it must go on the right of 4, and since 7 is larger than 6, it must go to the right of 6.
* When add an **Equal Value we are going to treat it as a Larger Value**.
* Say add 4 we would treat 4 as larger than 4 we can either send the node to the left or to the right, and that will be determined by whether we view it as being larger or smaller.
* We are going to view it as being larger.
* So, the 4 will go down to the 6, where it's determined that it's smaller than 6, so it will go to the left.
* We build up a Tree that satisfies the Binary Search Tree requirement of having the smallest value in the left-most node, and the largest value in the right-most node.





**Search for node whose value = 3. Finding (Root, 3) using the above logic**

* Start from the root node which is 4 which is the current node.
* current node = 4
* Is the current node 4 null --> False
* Is 4 = 3 -->False
* Is 3 < 4 --->True
* So Do the recursion again but now the new current node becomes the **LEFT node of current node (4)**
* New current node is **LEFT** of 4 which is 2
* current Node = 2
* So Do the recursion again with current node = 2
* Is current node (2) null ---> False
* Is current node (2) = 3 ---->False
* Is 3 < current node 3 < 2 --->False. It is **GREATER**
* So now DO recursion but the new current node becomes the **RIGHT node** of the current node (2)
* New current node is RIGHT of 2 which 3.
* current node = 3. So DO recursion again
* Is current node (3) null ---> False
* Is current node (3) = 3 ---->True Return current node 3.
* Search for node whose value = 8 which is not present in the tree. Finding (Root, 8)

**Search for node which does not exist in the tree whose value = 8. Finding (Root, 8) using the above logic**

* Start from the root node which is 4 which is the current node. current node = 4
* Is the current node 4 null --> False
* Is 4 = 8 -->False
* Is 8 < 4 --->False
* So Do the recursion again but now the new current node becomes the RIGHT node of current node (4)
* New current node is RIGHT of 4 which is 6
* current Node = 6 So Do the recursion again with current node = 2
* Is current node (6) null ---> False
* Is current node (6) = 8---->False
* Is 8 < current node 8 < 6 --->False. It is GREATER
* So now DO recursion but the new current node becomes the RIGHT node of the current node (6)
* New current node is right of 6 which 7.
* current node = 7. So DO recursion again
* Is current node (7) null ---> False
* Is current node (7) = 8 ---->False
* Is 8 < current node 8 < 7 --->False. It is GREATER
* So now DO recursion but the new current node becomes the RIGHT node of the current node (7)
* But there is NO node to the RIGHT of 7 it is null .SO current Node become = null
* current node = null. So DO recursion again
* Is current node null ---> True
* return null

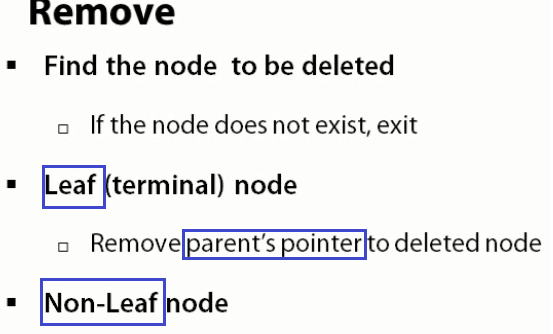
**Advantage of TREE over Linked List**

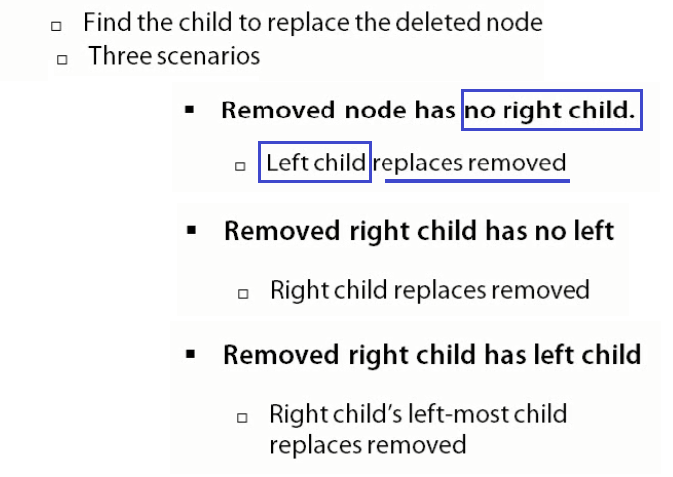
Node here we **did not have to look at every node** in the Tree to determine if a node exists or not.

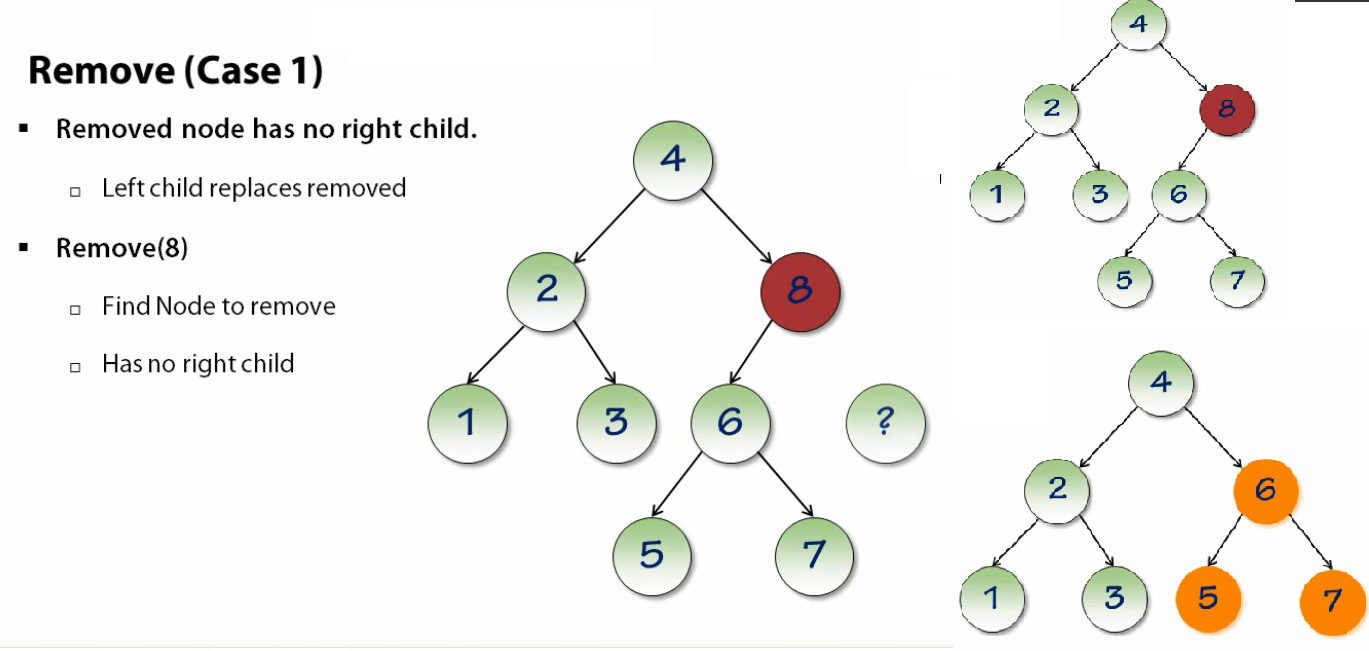
In a Linked List, the only way to determine that the value of a **node is NOT** in the Linked List would have been to have looked at every node in the list.

To check that nodes were or were not in the Tree while **looking at a subset** of the data in the Tree, and that's a very powerful mechanism as compared to a LIST where we need to look at all nodes in the list.

**REMOVING NODE FROM THE TREE**

****

****

****

**Removing a node that has no right child.**

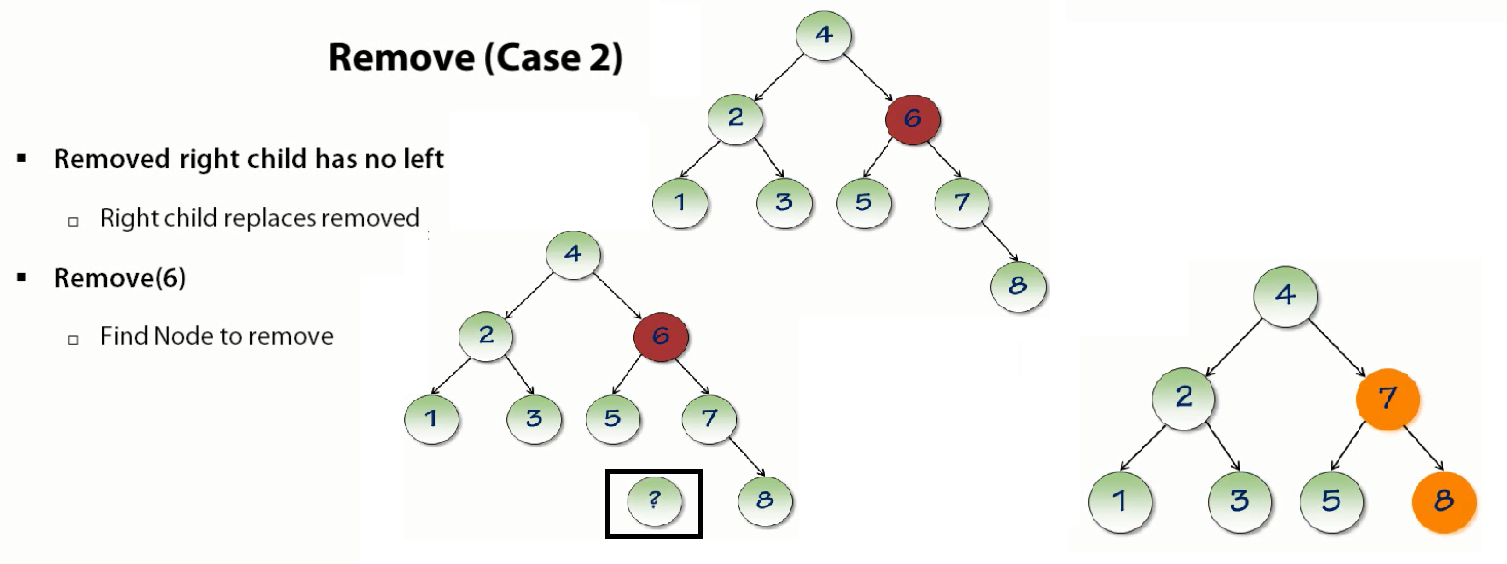
There are several nodes here that have no children to the right; the 8 node has no right child, and the 1, 3, 5, and 7 nodes have no right child.

So, this case would apply to the removal of any of those nodes.

We're going to look at the removal of 8.

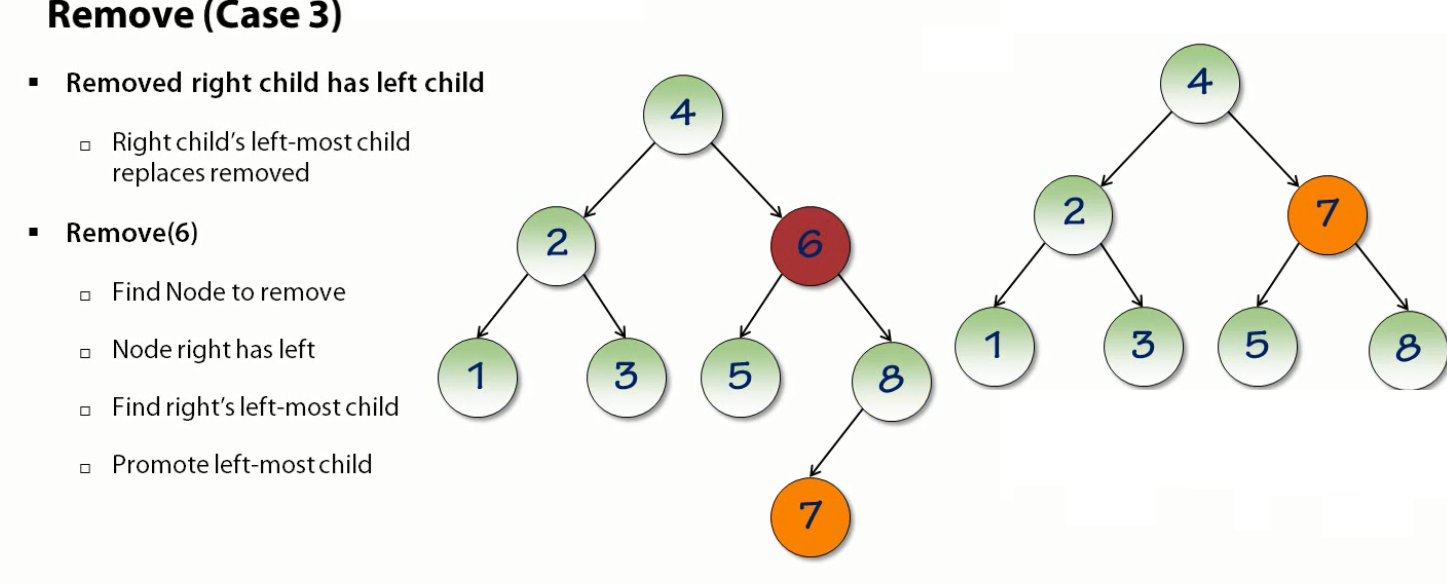
**Remove(8)**

* The first thing we do is we find the node to remove.
* The 4 is not it, 8 is larger so we end up on the 8 node.
* The 8 node has no right child.
* What we're going to do is promote its left child into its place.
* Since there's no child to the right, there is no value underneath the 4 that is going to be larger than 8.
* So, when we perform the delete operation, we've now promoted that Tree, the 6, 5, 7 nodes, up in place where the 8 was.
* This works because we haven't broken the invariant structure of the tree.
* Everything to the right of the Root Node 4 is greater than the Root Node,
* Everything to its left is smaller than 4, everything that's left is less than 4.
* So, we've been able to retain the rules

****

**Removing node has a right child that has no left child**

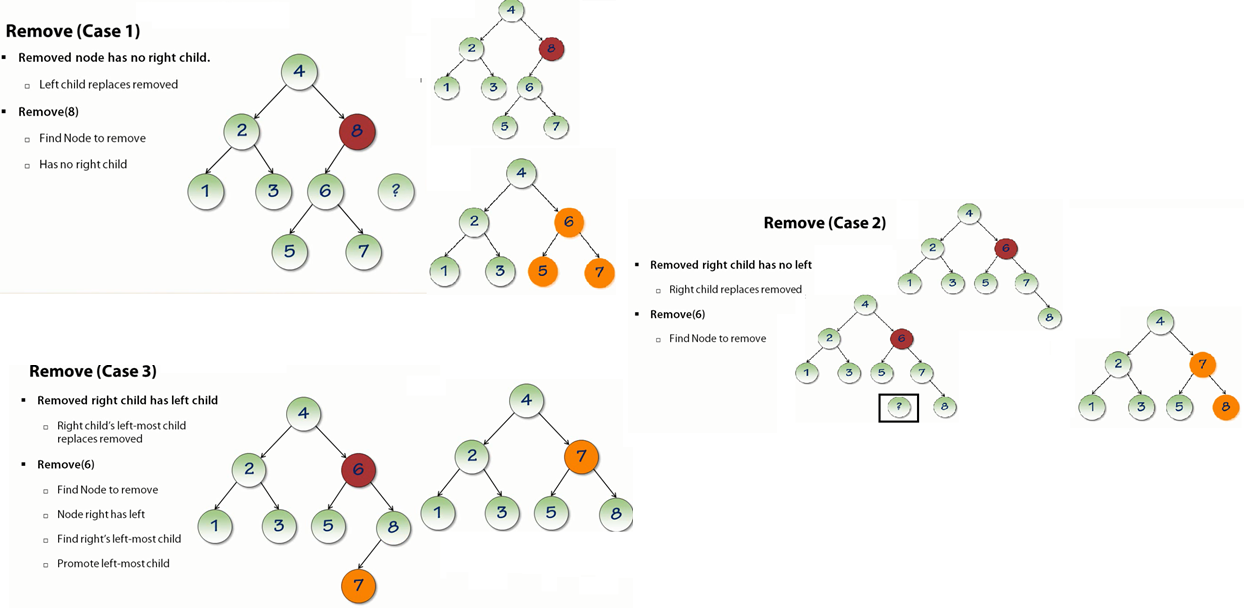
* Removing the node 6.
* The node 6 has a right child 7 that has **NO** left child.
* In this case the right child will replace the Removed node
* We're removing 6.
* We have to find the node first.
* It's greater than 4, so we end up on the 6 node.
* The removed node 6's right child (7) has no left child (? in figure) ie the 7 has no left
* So we're going to promote that right child 7 node, up to where the 6 node was.
* And now this is going to mean not just re-pointing 4 to 7, but 7 also has to become the parent of 5.
* So what we did there was we moved that entire right Tree up into the Removed node slot.
* We haven't broken the invariant structure of the tree.
* We know that everything to the right of 7 is going to be greater than 7, and everything to the left will be less than 7.

****

**Removed node's right child that has a left child.**

* Remove node 6 again.
* The 6 node has a right child 8 that has a left child 7
* Is the right child's left-most child node will replace the removed node(6).
* First we find the node 6 to remove.
* It's greater than 4, so we go to the right, and we've found 6.
* The node on the right 8 has a left child node 7.
* Find the right child's LEFT-MOST child.
* So, the right child is the 8, and its left-most child is 7.
* We promote the left most node (7) into the deleted slot that is the place where node 6 was present.
* Take 7 and move it up to where the 6 node was.
* We've retained the invariant structure.
* Everything greater than 7 is to the right, everything less than 7 is to the left.
* The reason we use the left-most child is because we know the left-most child is going to be the smallest value in the Tree.

**All the three cases of Removal summary**

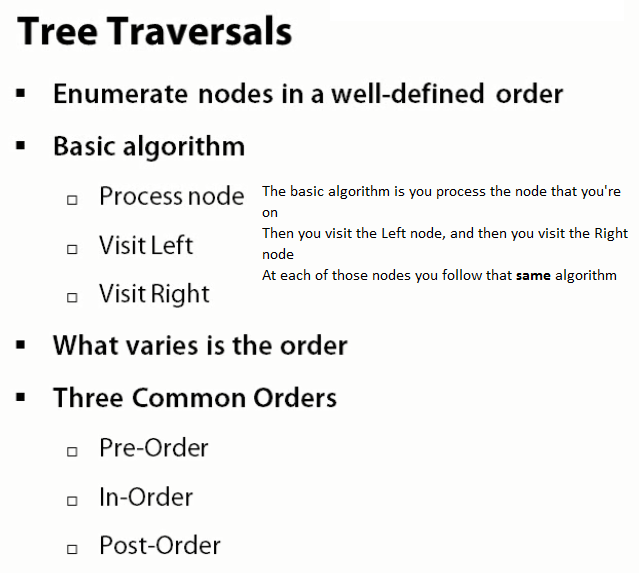
****

**TREE TRAVERSAL**

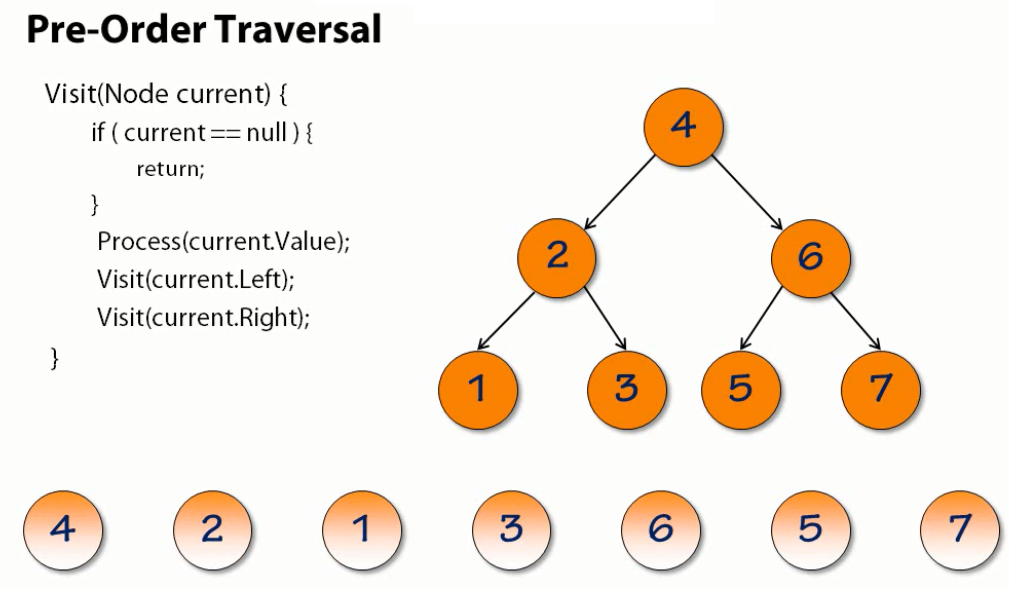
The basic algorithm is you process the node that you're on,

Then you visit the Left node, and then you visit the Right node,

At each of those nodes you follow that **same** algorithm



**PRE-ORDER TRAVERSAL**

****

* We're going to visit the Root Node of this Tree, and we want to traverse these items in Pre-Order.
* So, we start with the value 4; well 4 is not null.so we're going to Process that value.
* we've now processed the value 4, and I've indicated that along the bottom.
* Now we're going to visit the Left Node
* If the Left Node is not null we process it, which we did
* Let's go left again, and now we've processed it.
* We try to visit the Left Node, we will find it's **NULL** because there is no node to the left.
* We would now visit the Right Node, which is also **NULL** , and we return.
* So, now we're back at the 2 node, and we've already visited the left, so now let's visit the right.
* That causes us to process 3. Again, it has no children
* We return from 2, which is now processed left and right, so it returns back to 4.
* 4 is now going to process its right side.
* It starts with the node 6.
* The 6 node has a Left child 7, so it visits that;
* That 7 child has no left or right children so it returns,
* And the 6 node now processes its right node, which has no children, so we're done.
* So, a Pre-Order Traversal processed the nodes in the order 4, 2, 1, 3, 6, 5, 7.
* And this might look somewhat random to you, but what's important is that the enumeration order was stable.
* We could enumerate this Tree using Pre-Order Traversal a thousand times, and each time it will enumerate in this order.