Data Structure

- Queue Data Structure
- Stack Data Structure
- Binary Tree Traversal - BFS & DFS

Queue Data Structure

Introduction -

A Queue is a structure in which whatever goes first comes out first. In short FIFO - First In First Out.

In Queue insertion or addition in queue must happen from one end that is called rear or tail of the queue, and any removal or deletion must happen from front or head of the queue.

Operations of Queue -

- **enQueue(x) or push(x) (Insertion)**
- **deQueue() Or pop (Deletion)**
- **front() on peek()**
- **isEmpty()**

Push and Pop are more famous in context of stack whereas enQueue and deQueue is more famous in context of Queue.

All these operation must take O(1) time.

Implementation of Queue -

- Array Based Implementation
- LinkedList Based Implementation

```swift
// Array Implemantation

class Queue<T> {
    var arr = [T]()
    func enQueue(val: T) {
        arr.append(val)
    }
    func deQueue() -> T? {
        if arr.isEmpty {
            return nil
        } else {
            return arr.remove(at: 0)
        }
    }
}

var queue = Queue<String>()
queue.enQueue(val: "Hi")
queue.enQueue(val: "thr")
queue.deQueue()

// Linked List Implementation

class LLQueue<T> {
    var data: T
    var next: LLQueue?
    init(val: T) {
        data = val
    }
}

class LLQueueImp<T> {
    var head: LLQueue<T>?
    var front: LLQueue<T>? { return head }
    var rear: LLQueue<T>? {
        var node:LLQueue<T>? = self.head
        while (node?.next != nil) {
            node = node?.next
        }
        return node
    }

    func enQueue(val: T) {
        let newNode =  LLQueue(val: val)
        if let lastNode = rear {
            lastNode.next = newNode
        } else {
            head = newNode
        }
    }
```

```
    func deQueue() -> T? {
        if head == nil {
            return nil
        }
        let temp = head
        if head?.next != nil {
            head = head?.next
        } else {
            head = nil
        }
        return temp?.data
    }

    func peek() -> T? {
        return head?.data
    }

    func isEmpty() -> Bool {
        return (head == nil) ? true : false
    }
}


var llQueue = LLQueueImp<Int>()
llQueue.enQueue(val: 1)
llQueue.enQueue(val: 2)
llQueue.deQueue()
llQueue.peek()
```

Stack Data Structure

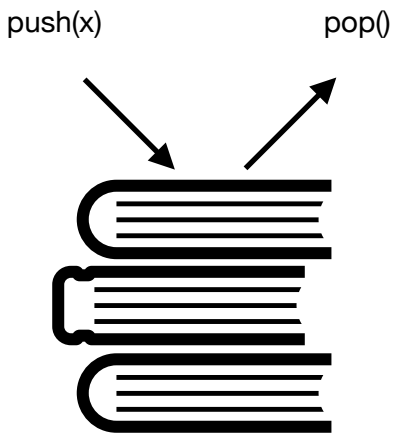
A stack is a structure in which whatever goes first comes out last. In short LIFO - Last In First Out.

In Stack insertion or addition, must happen from same end that is called top of a stack.

Operations -

- **push(x) (Insertion)**
- **pop (Deletion)**
- **peek()**
- **isEmpty()**

All these operation must take O(1) time.



Implementation of Stack -

- Array Based Implementation
- LinkedList Based Implementation

```swift
// Array Based Implementation
class Stack<Element> {

    var stackArray = [Element]()
    func  push(val: Element)  {
        stackArray.append(val)
    }
    func pop() -> Element? {
        return stackArray.removeLast()
    }
    func peek() -> Element? {
        return stackArray.last
    }
}


var stack = Stack<String>()
stack.push(val: "Hi")
stack.push(val: "This")
stack.push(val: "is")
stack.push(val: "Great")
stack.pop()
stack.peek()


// Linked List Based Implemention

class LLStack<T> {
    var value: T
    var next: LLStack?
    init(val: T) {
        value = val
    }
}

class LLStackImplementation<T> {
    var headNode: LLStack<T>?

    func pushInStack(val: T) {
        let newNode = LLStack(val: val)
        if headNode == nil {
            headNode = newNode
        } else {
            let oldHeadNode = headNode
            headNode = newNode
            newNode.next = oldHeadNode
        }
    }
```

```
    func popFromStack() -> T? {
        let currentTop = headNode
        if headNode?.next != nil {
            headNode = headNode?.next
        } else {
            headNode = nil
        }
        return currentTop?.value
    }

    func peek() -> T? {
        return headNode?.value
    }
}


var llstack = LLStackImplementation<String>()
llstack.pushInStack(val: "Hi")
llstack.pushInStack(val: "My")
llstack.pushInStack(val: "Name")
llstack.pushInStack(val: "is")
llstack.pushInStack(val: "Nobita")
llstack.popFromStack()
llstack.peek()
llstack.popFromStack()
```

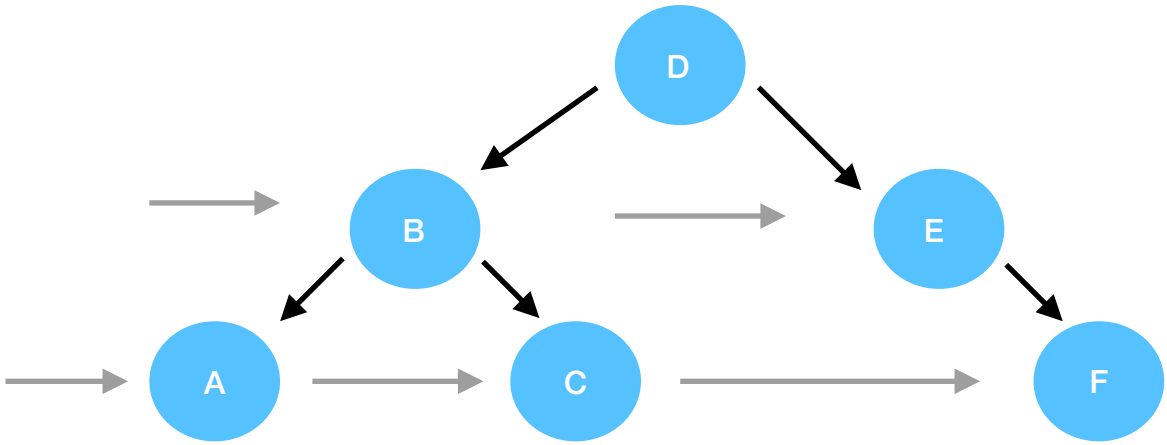
Binary Tree Traversal

Unlike other data structure like linked list or array, tree is not a linear data structure. In linked list which is a linear data structure, we always know that for each node or element we will have only one next element or node and we only need to traverse the tree in one direction.

Where as tree is not linear data structure, so for any node in tree we can have many possible direction or we can have many possible nodes.

Tree Traversal is the process of visiting each node in a tree in some order. There two main ways through which we traverse a tree -

- **Breadth first** - visit all node of same level and then go to next level



Breadth First Traversal - D -> B -> E -> A -> C -> F

Breadth First Or Level Order Traversal - We can use queue for this traversal. As we visit the node we can keep reference or address of all its children in a queue so that we can visit them later.

A node in a queue can be called discovered node whose address is known to us but we have not visited yet. Initially we can start with root node. We store the address of the root node in queue. As long as queue is not empty, we can take out the node and visit it and enqueue its children's, and we repeat this for all the nodes.

```swift
// Breadth First Traversal

class Node {
    var data: Int
    var leftNode: Node?
    var rightNode: Node?
    init(data: Int) {
        self.data = data
    }
}

func breadthFirstTraversal(root: Node) {
    let queue = LLQueueImp<Node>()
    queue.enQueue(val: root)
    while !(queue.isEmpty()) {
        let currentTop = queue.peek()
        print(currentTop?.data ?? 0)
        if let left = root.leftNode {
            queue.enQueue(val: left)
        }
        if let right = root.rightNode {
            queue.enQueue(val: right)
        }
        queue.deQueue()
    }
}
```

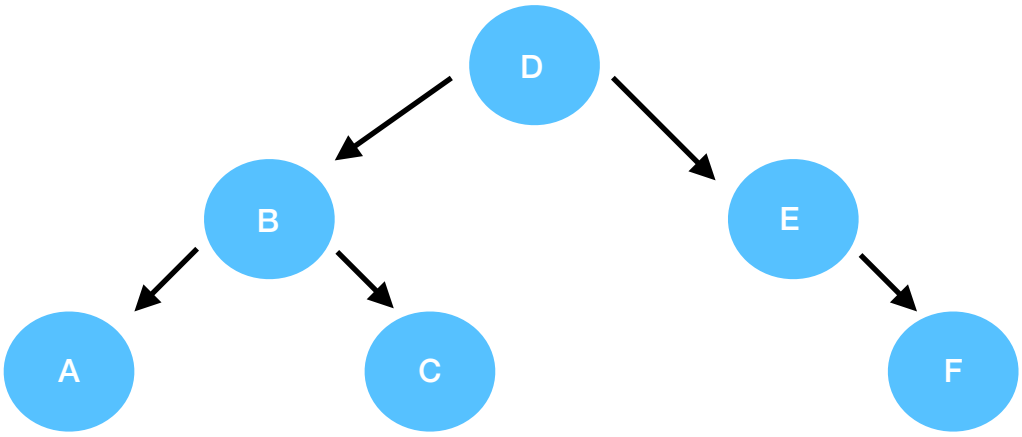
Space Complexity - O(h) // h is height of tree
best case - O(1)
worst/avg case - O(n)

Time Complexity - O(n)

- **Depth first -** visit all the node of child node before going to any next node

Depth First Traversal - we have three approach -

- Preorder Traversal - Root -> Left -> Right
- In-order Traversal - Left -> Root -> Right
- Postorder Traversal - Left -> Root -> Right



Pre Order Traversal - Root -> Left -> Right
D -> B -> A -> C -> E -> F

In Order Traversal - Left -> Root -> Right
A -> B -> C -> D -> E -> F

Post Order Traversal - Left -> Right -> Root
A -> C -> B -> F -> E -> D

Space Complexity - O(h) // h is height of tree
best/avg case - O(log n)
worst case - O(n)

Time Complexity - O(n)