Note - DS Trees are git-hub file-Name :- tree Intro & terminology

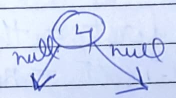
★ Data Structure Trees

★ Binary Tree

- Binary Tree node look like that :-

```
node {
    value : 4
    left : null
    right : null
}
```

→ it look like in graphically like that



★ Insertion at Binary Search Tree :-

(i) create newNode

(ii) if less < left
else ~~so~~ greater > right

(iii) if root node null insert newNode
else move to next

Steps :- to insert

(i) create newNode

(ii) let temp = this.root

(iii) while loop

(iv) if newNode === newNode return undefined

(v) if less (<) move to left
else (>) greater move to right

(vi) if root node null insert newNode
else move to next node.

```
const newNode = new Node(value);
~~let temp = this.root;~~
if (!this.root) return { this.root = newNode;
                         return this; }
let temp = this.root;
while (true){
    ~~if (this.temp.value === newNode) return~~
                    undefined
    if (temp.value === newNode) return
                    undefined
    if (temp.value > newNode.value){
        if (!temp.right){
            temp.right = newNode;
            return this;
        }
        temp = temp.right;
    }else{
        if (!temp.left){
            temp.left = newNode;
            return this;
        }
        temp = temp.left;
    }
}
}
```

☆ Search an Node in (BST)

- Step to Search An Node :-

(i) if root ===null return false
(ii) let temp = this.root;
(iii) while (temp){
    if < left
    else if > right
    else = return true
return false

```
if (!this.root) return false;
  let temp = this.root;
  while(temp){
    if (value > temp.value){
        temp=temp.right;
    }
    else if (value < temp.Value){
        temp= temp.left;
    } else
        return true
    }
}
```

☆ Method to find minimum value in Node of (BST)

```
minValNode (currNode){
  while (currNode.left != Null){
      currNode = currenNode.left;
  } return currenNode
```

☆ Hash Tables

☆ Constructor of Hash tables :-

```
Class HashTables {
    Constructor (size=7){
        this.dataMap = new Array (size)
    }
```

☆ Hash Function

```
_hash (Key){
    let hash=0;
    for (let i=0; i<Key.length; i++){
    hash =(hash +key.charCodeAt(i) *23)%
                this.dataMap.length;
    }
    return hash;
```

☆ Set method of Value-Key pair :-

```
set (Key, value){
    let index =this._hash (key);
    if (!this.dataMap[index]){
        this.dataMap[index]=[];
    }
    this.dataMap[index].push([Key, value]);
    return this;
```

☆ Get method in Hash table

```
get (key){
  let index = this._hash(key);
  if (this.dataMap[index]){
      for (let i=0; i < this.dataMap[index].length; i++){
        if (this.dataMap[index][i][0] ===key){
          return this.dataMap[index][i][1];
        }
      }
  }
  return undefined;
```

☆ Keys method in Hash tables

```
Keys (){
  let allKeys = [];
  for(let i = 0; i < this.dataMap.length; i++){
      if (this.dataMap[i]){
          for(let j = 0; j < this.dataMap[i][0]);
      }
  }
  return allKeys;
```

☆ Big O of Hash Tables

→ Big O of Set method is O(1)
→ Big O of get method is O(1)

☆ Data Structures Graphs

☆ Big O Graphs

(i) Graph Representations

- Adjacency Matrix
  - Space complexity : $O(V^2)$
  - checking if an edge exists : $O(1)$
  - Iterating over all edges : $O(V^2)$
  - adding vertex : $O(V^2)$
- Adjacency list :
  - space complexity : $O(V+E)$
  - checking if an edge exists : $O(V)$
  - Iterating over all edges : $O(V+E)$

☆ Constructor of graph

```
constructor(){
    this.adjacencyList = {};
}
```

☆ Print Graph () :-

```
if (object.keys(this.adjacencyList).length !==0){
  console.log("{");
  for (const [key, value] of object.entries(this.
                adjacencyList)){
      console.log(" " '${key}: ${value}');
  }console.log("}");
}else{
    console.log("{}");
```

☆ Adding Vertex to Graph

```
addVertex(vertex){
    if (!this.adjancecy[vertex]){
        this.adjancecy[vertex] = [];
        return true;
    }
    return false
}
```

☆ Adding ~~ad~~ ~~Egd~~ Edge in Graph:-

```
addEdge(vert1, vert2){
    if(this.adjancecyList[vert1] &&
        this.adjancecyList[vert2]){
    this.adjancecy[vert1].push(vert2);
    this.adjancecyList[vert2].push(vert1);
    return true;
    }
    return false;
```

☆ removeEdge from Graph:-

```
if (this.adjancecyList[vert1] && this.adjancecy
                                    [vert2]){
    this.adjancecy[vert1] = this.adjancecy[vert1].
                    filter(v => v != vert2)
    this.adjancecy[vert1] = this.adjancyList[vert1].
                    filter(v => v != vert1)
    return true
}
return ~~tru~~false;
```

☆ remove Vertex in Graph

```
removeVertex(vertex){
    if (!this.adjancecyList[vertex]) return undefined
    while (this.adjancecyList[vertex].length){
        let temp = this.adjacancecy[vertex].pop();
        this.removeEdge(vertex, temp);
    }
    delete this.adjancecyList[vertex];
    return this;
```

☆ Recursion

A function that calls itself... until it doesn't. It particularly useful for problems that can be broken down into smaller, similar sub problem.

☆ Key concept

- **Base case** - This is the condition under ~~function~~ which the recursive function calling itself. It prevents infinite recursion and ensures that the function eventually terminates.
- **Recursive case** - This is where the function calls itself with a modified argument, gradually approaching the base case.

☆ Call Stack Example

```
function funcThree(){
        console.log("Three");
}

function funcTwo(){
        ~~console.log(~~
        funcThree();
    console.log("Two");
}

function funcOne(){
        funcTwo();
    console.log("One");
}

funcOne(); // Three, Two, One
```

☆ Basic Sort

(i) Bubble Sort

Bubble sort is simple comparision based sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing each pair adjacent items and swapping them if they are in wrong order.

→ This process is repeated until the list is sorted.

```
let temp
for (let i = 0; i < arr.length; i++){
    for(let j=0; j < arr.length-i-1; j++){
        if (arr[j] > arr[j+1]){
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = ~~arr~~[temp];
        }
    }
}
return arr;
```

☆ Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It work by dividing the input list into two parts: a sorted sublist of item which is built up from left to right at the front (left) of the list and a sublist of ~~item~~ the remaining unsorted items. The algorithm proceeds by find the smallest (or largest, depending on sorting order) element from the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element, and moving sublist boundaries one element to the right.

```
let min}
for (i=0; i <arr.length-1; i++){
    min =i
    for(j =i+1; j <arr.length; j++){
        if (arr[j] < arr[min]){
            min =j
        }
    }
    if (i !=min){
      let temp = arr[j]
      arr[j] = arr[min]
      arr[min] = temp
    }
}
return arr
}
```

☆ Insertion Sort

Insertion sort is another simple comparison based sorting algorithm. It build the final sorted array one item at a time, with the benefit of being much more efficient in practice than other simple quadratic algorithms such as selection sort or bubble sort.

```
for (let i=1 ; i<arr.length; i++){
    let key = arr[i];
    let j = i - 1;
    while ( j >= 0 && arr[j] > key){
        arr[j+1] = arr[j]
        j - -;
    }
    arr[j+1] = key;
}
return arr
```

☆ Time complexity :

- Worst - case : $O(n^2)$ when the array is sorted in reverse order.
- Average - case : $O(n^2)$
- Best case : $O(n)$ when the array is already sorted.

☆ Space complexity : $O(1)$ because it is an in sorting algorithm.