Data Structure Workouts

1. *Learn the concepts of Tree. Complete at least three sample workouts.*
2. *Learn the concepts of Binary Search Tree. Complete at least three sample workouts.*
   Example:
   a. *Create a Binary Search Tree with insertion, contains, delete, three traversals (postorder, preorder, in order).*
   b. *Find the closest value to a given number in a Tree.*
   c. *Validate whether a given tree is BST or not.*
3. *Learn the concepts of Heap. Complete at least three sample workouts.*
   Example:
   a. *Create a min heap & max heap with build, insert, remove.*
4. *Learn the concept of Heap sort. Complete at least three sample workouts*
5. *Learn the concepts of Trie. Complete at least 3 sample workouts.*
6. *Learn the concepts of Graph. Complete at least three sample workouts.*
7. *Learn the concepts of Graph traversals (BFS, DFS).*
8. *Do at least 3 problems each for every structure from any competitive coding websites*
9. *Learn about the applications of all structures you covered this week*

Write a short description about this task

Link to the folder containing code and screenshot of the output

Tree:

A Tree is a hierarchical data structure that is used to represent and organise data in a way that is easy to navigate and search it is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes

The top most node of the tree is called root and the nodes below it are called child node

Each node cab also have their own child node forming a recursive structure

Tree usage:

1.File system for directory structure

2.DOM

3.Chat bots

4 Abstract syntax trees

5. A family tree

6.An organisation tree

Basic Terminologies in a Tree

1,Parent node:The node which is the immediate predecessor of the node is called a parent node

2,node: The node which is the immediate succesor of node is called child node

3,Root node: The top most node of a tree which does not have any parent node are called root node

4,Leaf node or External node: The nodes which does not have any child nodes are called leaf node

5,Ancestor of a node: Any predecessor node on te path of the root to that node are called ancestor of that node

6,Descendant:Any successor node in the path from the leaf node are the descendants of node B

7,Sibling:Children of the same parent node are called siblings

8,level of node: the count of edges on the path from the root node to that node .the root node has level 0

9,Internal node: A node with at least one child is called internal node

10.Neighbour if a node: Parent or child nodes of that node are called neighbours of that node

11,subtree:Any node of a tree along with its descendant

Properties of a tree:

Number of edges:

An edge can be defined as a connection between two nodes .if a tree has n nodes then it will have n-1 edges .there is only one path from each node to any other node of the tree

Depth of a node: Depth of a node is defined as the length of path from root to that node.Each edge adds 1 unit of length to the path so it can also be number of edges in the path from root of the tree to the node

Height of node: length of the longest path from node to the leaf of the tree

Height of tree: length of the longest path from root of tree to the leaf of the tree

Degree if a Node: The total count of subtrees attached to that node is called the degree of the node .the degree if the leaf node must be zero

Degree of tree: is the maximum degree of a node among all nodes in the tree

Types of tree Data structure

Binary Tree:

A Binary tree is a tree in which each parent node at most has two children

Binary Search Tree

Bst is a tree ds in which each node has a maximum of two children

All nodes of the left subtree are less than the root node

All node of the right subtree are more than the root node

AVL Tree

AVL tree is a self balancing tree in which each node maintains a balance factor whose value is either -1 0 and 1

BAlance factor=(Height of the left subtree)-(Height of the right subtree)

Or

(Height of the right subtree)-(Height of the left subtree)

Btree

ALso known as height balanced m-way tree

Special kind of self balancing search tree in wich each node can contain more than one key and can have more than two children

Balanced Tree

If the height of *left sub tree abd right subtree is equal or differs utmost by 1 .thr tree is known as balanced tree*

Tree Traversal

1.in order: left root right

2.preorder: root left right

3.postorder : left right root

Write a short description about this task

Link to the folder containing code and screenshot of the output

Write a short description about this task

Link to the folder containing code and screenshot of the output

A Heap is a specialised tree-based data structure that satisfies the heap property. The heap property specifies that for each node in the heap, the value stored in that node is greater than or equal to (in a max heap) or less than or equal to (in a min heap) the values stored in its children. A heap is typically implemented as an array, where the root of the heap is stored in the first position (index 0) and the children of a node at index i are stored at indices 2i+1 and 2i+2. This layout allows for efficient insertion and removal of elements, as well as finding the minimum or maximum element in constant time.
Heaps are commonly used in algorithms that require efficient priority queue operations, such as Dijkstra's algorithm for finding the shortest path in a graph. They are also used in heap sort, a popular sorting algorithm that uses a heap to sort elements in ascending or descending order.

```
function heapsort(arr)
{
    let n=arr.length
    for(i=Math.floor(n/2)-1;i>=0;i--)
    {
        heapify(arr,n,i)
    }
    for(i=n-1;i>0;i--)
    {
        let temp=arr[0]
        arr[0]=arr[i]
        arr[i]=temp

        heapify(arr,i,0)
    }
    return arr


}




function heapify(arr,n,i)
{
    let largest=i
    let left=2*i+1
    let right=2*i+2

    if(left<n && arr[left]>arr[largest])
```

```
    {
        largest=left
    }
    if(right<n &&arr[right]>arr[largest])
    {
        largest=right
    }
    if(largest!==i)
    {
        let temp=arr[i]
        arr[i]= arr[largest]
        arr[largest]=temp
    }
}

const arr=[8,2,-1,5,7,3,4]
const res=heapsort(arr)
console.log(res)
```

Write a short description about this task

Link to the folder containing code and screenshot of the output

```
class Heap {
    constructor() {
      this.heap = [];
    }

    buildHeap(arr) {
      this.heap = arr;
      for (let i = Math.floor((this.heap.length-2)/2); i >= 0; i--) {
        this._heapifyDown(i);
      }
    }

    insert(value) {
      this.heap.push(value);
      this._heapifyUp(this.heap.length-1);
    }

    remove() {
```

```
    if (this.heap.length <= 0) {
      return null;
    }
    if (this.heap.length === 1) {
      return this.heap.pop();
    }
    const root = this.heap[0];
    this.heap[0] = this.heap.pop();
    this._heapifyDown(0);
    return root;
  }

  _heapifyUp(index) {
    let parent = Math.floor((index-1)/2);
    while (index > 0 && this.heap[parent] < this.heap[index]) {
      const temp = this.heap[parent];
      this.heap[parent] = this.heap[index];
      this.heap[index] = temp;
      index = parent;
      parent = Math.floor((index-1)/2);
    }
  }

  _heapifyDown(index) {
    let left = 2*index + 1;
    let right = 2*index + 2;
    let largest = index;
    if (left < this.heap.length && this.heap[left] >
this.heap[largest]) {
      largest = left;
    }
    if (right < this.heap.length && this.heap[right] >
this.heap[largest]) {
      largest = right;
    }
    if (largest !== index) {
      const temp = this.heap[index];
      this.heap[index] = this.heap[largest];
      this.heap[largest] = temp;
      this._heapifyDown(largest);
    }
  }
```

```
  }
  const maxHeap = new Heap();

maxHeap.insert(4);
maxHeap.insert(1);
maxHeap.insert(7);
maxHeap.insert(3);

console.log(maxHeap.heap); // [7, 4, 1, 3]


maxHeap.remove();

console.log(maxHeap.heap); // [4, 3, 1]
```

```
class Heap {
    constructor() {
      this.heap = [];
    }

    buildHeap(arr) {
      this.heap = arr;
      for (let i = Math.floor((this.heap.length-2)/2); i >= 0; i--) {
        this._heapifyDown(i);
      }
    }

    insert(value) {
      this.heap.push(value);
      this._heapifyUp(this.heap.length-1);
    }

    remove() {
      if (this.heap.length <= 0) {
        return null;
      }
      if (this.heap.length === 1) {
        return this.heap.pop();
```

```
    }
    const root = this.heap[0];
    this.heap[0] = this.heap.pop();
    this._heapifyDown(0);
    return root;
  }


    _heapifyUp(index) {
        let parent = Math.floor((index-1)/2);
        while (index > 0 && this.heap[parent] > this.heap[index]) {
          const temp = this.heap[parent];
          this.heap[parent] = this.heap[index];
          this.heap[index] = temp;
          index = parent;
          parent = Math.floor((index-1)/2);
        }
      }

  _heapifyDown(index) {
    let left = 2*index + 1;
    let right = 2*index + 2;
    let smallest = index;
    if (left < this.heap.length && this.heap[left] <
 this.heap[smallest]) {
      smallest = left;
    }
    if (right < this.heap.length && this.heap[right] <
 this.heap[smallest]) {
      smallest = right;
    }
    if (smallest !== index) {
      const temp = this.heap[index];
      this.heap[index] = this.heap[smallest];
      this.heap[smallest] = temp;
      this._heapifyDown(smallest);
    }
  }
}


const minHeap = new Heap();
```

```
minHeap.insert(4);
minHeap.insert(1);
minHeap.insert(7);
minHeap.insert(3);

console.log(minHeap.heap); // [1, 3, 4, 7]

minHeap.remove();

console.log(minHeap.heap); // [3, 4, 7]
```

Write a short description about this task

Link to the folder containing code and screenshot of the output

Trie Data Structure:

Trie is a type of k-ary search tree used for storing and searching a specific key from a set. Using Trie, search complexities can be brought to optimal limit (key length).

A trie (derived from retrieval) is a multiway tree data structure used for storing strings over an alphabet. It is used to store a large amount of strings. The pattern matching can be done efficiently using tries.

```
// TrieNode class in JS
class TrieNode
{
        constructor()
        {
                this.children = new Array(26)

                // isEndOfWord is True if node represent the end of the word
                self.isEndOfWord = false;

        }
}

// This code is contributed by phasing17.
```

Advantages of tries

1. In tries the keys are searched using common prefixes. Hence it is faster. The lookup of keys depends upon the height in case of binary search tree.
2. Tries take less space when they contain a large number of short strings. As nodes are shared between the keys.
3. Tries help with longest prefix matching, when we want to find the key.

Comparison of tries with hash table

1. Looking up data in a trie is faster in worst case as compared to imperfect hash table.
2. There are no collisions of different keys in a trie.
3. In trie if single key is associated with more than one value then it resembles buckets in hash table.
4. There is no hash function in trie.
5. Sometimes data retrieval from tries is very much slower than hashing.
6. Representation of keys a string is complex. For example, representing floating point numbers using strings is really complicated in tries.
7. Tries always take more space than hash tables.
8. Tries are not available in programming tool it. Hence implementation of tries has to be done from scratch.

Applications of tries

1. Tries has an ability to insert, delete or search for the entries. Hence they are used in building dictionaries such as entries for telephone numbers, English words.
2. Tries are also used in spell-checking softwares.

Write a short description about this task

Link to the folder containing code and screenshot of the output

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).

- ***Vertices:*** *Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labelled or unlabelled.*
- ***Edges:*** *Edges are drawn or used to connect two nodes of the graph. It can be ordered by a pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabeled*

Applications

Google map,Social media websites,Networks

Write a short description about this task

Link to the folder containing code and screenshot of the output

```
// Javacript Program to print BFS traversal from a given
    // source vertex. BFS(int s) traverses vertices
    // reachable from s.


    // This class represents a directed graph using
    // adjacency list representation
    class Graph
    {

        // Constructor
        constructor(v)
        {
            this.V = v;
            this.adj = new Array(v);
            for(let i = 0; i < v; i++)
                this.adj[i] = [];
        }

        // Function to add an edge into the graph
        addEdge(v, w)
        {

            // Add w to v's list.
            this.adj[v].push(w);
        }
```

```
    // prints BFS traversal from a given source s
    BFS(s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        let visited = new Array(this.V);
        for(let i = 0; i < this.V; i++)
            visited[i] = false;

        // Create a queue for BFS
        let queue=[];

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.push(s);

        while(queue.length>0)
        {
            // Dequeue a vertex from queue and print it
            s = queue[0];
            console.log(s+" ");
            queue.shift();

            // Get all adjacent vertices of the dequeued
            // vertex s. If a adjacent has not been visited,
            // then mark it visited and enqueue it
            this.adj[s].forEach((adjacent,i) => {
                if(!visited[adjacent])
                {
                    visited[adjacent]=true;
                    queue.push(adjacent);
                }
            });
        }
    }
}

// Driver program to test methods of graph class

// Create a graph given in the above diagram
g = new Graph(4);
g.addEdge(0, 1);
```

```
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    console.log("Following is Breadth First Traversal " +
                "(starting from vertex 2) ");

    g.BFS(2);

    // This code is contributed by Aman Kumar.
```

Write a short description about this task

Link to the folder containing code and screenshot of the output

```
// JS code to implement the approach

// Function to find the level of the given node
function findLevel( N, edges, X)
{
    // Variable to store maximum vertex of graph
    let maxVertex = 0;
    for (let i=0;i<edges.length;i++){
        let it = edges[i];
        let a = Math.max(it[0],it[1]);
        maxVertex = Math.max(maxVertex,a);
    }

    // Creating adjacency list
    let adj = [];
    for(let i=0;i<maxVertex+1;i++){
        adj.push([]);
    }
    for (let i = 0; i < edges.length; i++) {
        adj[edges[i][0]].push(edges[i][1]);
        adj[edges[i][1]].push(edges[i][0]);
    }

    // If X is not present then return -1
```

```
    if (X > maxVertex || adj[X].length == 0)
        return -1;

    // Initialize a Queue for BFS traversal
    let q = [];
    q.push(0);
    let level = 0;

    // Visited array to mark the already visited nodes
    let visited = [];
    for(let i=0;i<maxVertex+1;i++)
    {
        visited.push(0);
    }
    visited[0] = 1;

    // BFS traversal

    while (q.length > 0) {
        let sz = q.length;
        while (sz--) {
            let currentNode = q[0];
            q.shift();
            if (currentNode == X) {
                return level;
            }

            for(let k =0;k<adj[currentNode].length;k++){
                let it = adj[currentNode][k];
                if (visited[it]==0) {
                    q.push(it);
                    visited[it] = 1;
                }
            }
        }
        level++;
    }

    return -1;
}

// Driver Code
```

```
let V = 5;
let edges
    = [ [ 0, 1 ], [ 0, 2 ], [ 1, 3 ], [ 2, 4 ] ];
let X = 3;

// Function call
let level = findLevel(V, edges, X);
console.log(level);

// This code is contributed by ksam24000
```

Write a short description about this task

Heap Data Structure

A heap is a complete binary tree where each node is greater than or equal to (in a max heap) or less than or equal to (in a min heap) its children. Heaps are commonly used to implement priority queues, which are data structures that allow efficient access to the element with the highest (or lowest) priority. Some common applications of heaps include:

- *Dijkstra's shortest path algorithm*
- *Huffman coding for data compression*
- *Median-finding algorithms*
- *Event-driven simulations*

Tree Data Structure

A tree is a collection of nodes connected by edges that has a single root node and no cycles. Trees are widely used in computer science for various applications, such as:

- *Representing hierarchical relationships (e.g., file systems, organization charts)*
- *Implementing search algorithms (e.g., binary search trees, AVL trees, red-black trees)*
- *Parsing and processing data (e.g., syntax trees for programming languages)*

- *Storing data for efficient access and retrieval (e.g., B-trees)*

Trie Data Structure

A trie (pronounced "try") is a tree-like data structure that stores a set of strings or sequences. Each node in a trie represents a character or a prefix of a string, and the edges between nodes are labeled with characters. Tries are commonly used for:

- *Text and pattern matching (e.g., autocomplete suggestions, spell-checking)*
- *IP routing and network protocols (e.g., routing tables, DNS)*
- *Storing dictionaries and other sets of words*

Graph Data Structure

A graph is a collection of nodes (also called vertices) connected by edges. Graphs can be directed or undirected, weighted or unweighted, and may have cycles or not. Graphs are used to model various types of relationships, such as:

- *Social networks (e.g., Facebook, Twitter)*
- *Transportation networks (e.g., road networks, airline routes)*
- *Electrical circuits and circuit diagrams*
-
-
-
- *Recommendation systems and collaborative filtering*
- *a spanning tree of a graph is a way to connect all the vertices in the graph without creating any loops or cycles. It's like taking the original graph and simplifying it by removing some edges, but still keeping all the vertices connected. There can be many different spanning trees for the same graph, but they all have the same set of vertices and no cycles. Spanning trees have many uses, like finding the shortest path between two points in a graph or designing computer networks.*