**✅ Session Ground Rules**

**🧑‍🏫 Learning & Participation**

1. **Be Present:** Please join on time and stay for the full session.
2. **Stay Engaged:** Participate actively — ask questions, share thoughts, and collaborate **when prompted.**
3. **Use the Chat Wisely:** Drop questions, but avoid spamming or unrelated messages.

**🎧 Tech Etiquette**

1. **Mute When Not Speaking:** Keep your mic muted unless you’re asking or answering something.
2. **Camera Optional but Encouraged:** If you're comfortable, keep it on — it helps build connection.

**📚 Content & Recording**

1. **No Unauthorized Recording:** Please don’t record or share the session without permission.
2. **Materials Access:** Any slides, code, or recordings (if applicable) will be shared after class.

**🤝 Respect & Inclusion**

1. **Respect All Opinions:** There are no “silly” questions — we’re all here to learn.
2. **Keep It Professional:** No offensive, disrespectful, or disruptive behavior will be tolerated.

**🚨 Support & Issues**

1. **Facing Tech Issues?** Let me or the moderator know in the chat — we’ll try to help immediately.
2. **Missed Something?** Don’t worry, you’ll get resources to review.

**💡 Final Thought:**

“Learning is a team sport — **let's support each other and make the most of our time together**!”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Phase 1** | **JFSD: Data Structures and Algorithms** | | |
| 1 | 6/7/2025 |  | 4 | Foundations of Data structures and Algorithms (Time and space complexity), Arrays, Multidimentional array , |
| 2 | 6/8/2025 |  | 4 | LinkedList, Operations on LinkedList, Stacks, Queue |
| 3 | 6/14/2025 |  | 4 | Trees Binary, AVL tree implemenation , Graphs and its types (Graph traversal implementation |
| 4 | 6/15/2025 |  | 4 | HashMap Bubble sort and Selection sort implementation Insertion sort, merge sort, quick sort implementation |
| 5 | 6/21/2025 |  | 4 | Heap sort, count sort,  Linear search, Binary Search, Jump Search algorithm |
| 6 | 6/22/2025 |  | 4 | radix sort implementation Phase end project |

Phase 1:

------------------------------------------------------------------------------------------------------------------------------------------  
**21-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**Merge Sort:**

Merge Sort is a classic **Divide and Conquer** algorithm that works by,

1. Dividing the array into halves until each subarray has only one element
2. Merging those sorted halves back together in order.

[2] [1]

[1,2]

[38] [27,43]

[27, 38, 43]

~~[27,~~**~~38~~**~~,43]~~ ~~[3~~**~~,~~**~~9,10~~,**82, 99,100,102**]

[3, 9 , 10, 27, 38, 43, 82]

**Quick Sort**

Quick Sort is also a Divide and Conquer algorithm.

Unlike merge sort, which divides array in half, quicksort works by selecting a **“pivot”** element, then partitions the array so:

* All elements < pivot go to the left
* All elements > pivot go to the right

Then recursively softs the left and right subarrays.

Binary Search:

Idea: Cut the sorted array in half repeatedly until element is found.

* **Use when the array is sorted**
* You want fast performance (much faster than linear search)

Target=11

[1,3,5,7,9,11]

16

[1,3**,**5,7,9,11,13,23,25,28,40,45,46,53,57,67,**68**,69,72,**89**,100]

i=1

while((i<n) && arr[i]<target) {

i=i\*2;

}

I=32

------------------------------------------------------------------------------------------------------------------------------------------  
**15-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**HashMap**

A hashMap is a key-value pair datastructure

Fast insertion, searching and deletion operations

Uses Hash Function to map keys to indices in array (buckets)

Name – Dhruvik

Name – Mohit

Age= 25

Characteristics:

* Key-Value pairs –{ name – Dhruvik, }
* No Duplicate keys allowed
* Allow Null Values as key and value
* Unordered – Doesn’t maintain insertion order
* Efficient lookup – O(1)

------------------------------------------------------------------------------------------------------------------------------------------  
**14-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**What is a Non-Linear Data Structure?**

Data elements are not arranged sequentially (not in straight line). Instead, they form a hierarchy or relationship between elements (like tree or graph)

* Elements can have multiple relations
* Not stored in contiguous memory locations
* Traversal is complex (Compared to array or linked list)

Examples:

* Tree
* Graph
* Heap

Why Non-Linear?

* To manage relationships (e.g. file system, organization charts)
* For efficient search and insert operations

**Tree**

A Tree is a hierarchical, non-linear datastructure made up of nodes

* Top node – root
* Each node (except root) has one parent
* Nodes may have children
* There are no cycle

A level 0

B C level 1

C E F

Use case?

* File system hierarchy
* XML/HTML DOM
* Databases (B-Tree, B+Tree)
* Routing algorithms in networks

Advantage

* Dynamic insertion/deletions
* Fast searching with ordered tree
* Mirrors natural hierarchical relationships

Terminologies:

* Node – Basic unit of tree with data
* Root Node – The topmost node
* Parent Node – A node that has children
* Child node – Node descended from another node
* Leaf node – A node with no children
* Internal Node – A node with atleast one child
* Degree – Number of children of a node
* Level – Distance from root (root – level 0)
* Height of Node – Longest path from node to a leaf
* Depth of node – Path length from root to that node

Types of tree:

* Binary Tree
* BST Tree
* AVL Tree
* B Tree

**Binary Tree**

* Each node has at most two children, (0,1,2)
* Foundation of more advanced trees like Binary Search Trees, AVL Trees, Heaps etc

Class TreeNode {

int data;

TreeNode left;

TreeNode right;

}

**Types of Binary Trees**

1. **Full Binary Tree –** Every node has 0 or 2 children
2. **Perfect Binary Tree** 
   1. **All interior nodes have two children**
   2. **All leaves at the same level**

**1**

**2 3**

**4 5 6 7**

1. **Complete Binary Tree**

All levels are fully filled except last, and the last level is filled left to right.

1. **Skewed Binary Tree**

All nodes have only one child (either left or right)

**Binary Search Tree**

A Binary Search tree is binary tree with a special properties

* All nodes in the left subtree have values less than node’s value
* All the nodes in the right subtree have values greater than node’s value
* This rule applies recursively to all nodes

**AVL Tree**

An AVL Tree is self-balancing Binary Search Tree (BST) where the difference in heights between left and right subtree of any node is at most 1.

The height difference is called Balance Factor –

Balance Factor = Height(left\_subtree) – height(right\_subtree)

Valid values of BF - -1,0,1

How ?

LL – Left Left - Inserted into left of left

RR – Right Right

LR – Left Right

RL – Right Left

**B- Tree**

A B-Tree is a self-balancing search tree that is optimized for systems that read and write large blocks of data like datasets and file systems

Generalization of BST but allows a node to have more than two childrens

Fundamentals:

1. Multi-way tree
   * A B-Tree of order m, can have upto m children per node
   * Each node can hold multiple keys (upto m-1 keys) arranged in **sorted order**
2. Balanced Tree
   * The B-Tree is always balanced – all leafs nodes are at the same level
   * It keeps height small for fast access.
3. Sorted Keys
   * Keys in each node are kept in sorted order
   * Search is done using binary search within the node
4. Broad and shallow

**Tree Traversals**

1. Depth First Traversal (DFT)
   1. Inorder (Left -> root -> right)
   2. Preorder (Root -> left -> right)
   3. Postorder (Left -> right -> root)
2. Breadth First Traversal (BFT)

**Graph**

Graph is a non linear datastructure consisting of:

Vertices (or nodes) – Points

Edge (or arcs) – connection between Vertices

Real-world examples:

* Social network (people – nodes, relationships – edge)
* Maps (Cities – nodes, roads – edge)
* Internet (Web pages = nodes, hyperlinks- edge)

Treee and Graph?

|  |  |  |
| --- | --- | --- |
| Feature | Tree | Graph |
| Structure | Hierarchical | Network-like |
| Cycles | No cycles | May have cycle |
| Parent/Child | Defined | No such concept |
| Connectivity | One path between nodes | Multiple paths may exist |
| Edge Count | n-1 edges | Can have any number of edges |
| Root | Always has one root | No root required |
| Direction | Usually one direction | Can be directed/undirected |

|  |  |  |
| --- | --- | --- |
| Term | Meaning | |
| Vertex(Node) | | Fundamental unit of graph |
| Edge(Arc) | | Connection between two nodes |
| Adjacency | | Two nodes are adjacent if they are connected by an edge |
| Path | | A sequence of vertices connected by edges |
| Degree | | Number of edges connected to a node |
| Cycle | | A path that starts and ends at the same vertex |
|  | |  |

**Types of graphs**

1. **Based on Structure**
   1. Finite Graph
      1. Has finite number of nodes and edges
   2. Infinite Graph
      1. Theoretically contains infinite nodes/edges
      2. Not practical in memory, used in theoretical math
   3. Trivial Graph
      1. Has only one vertex and **no edges**
   4. Null Graph
      1. Contains n vertices and no edges
   5. Simple Graph
      1. No Self-loops or multiple edges
   6. Multi Graph
      1. Has multiple edges (parallel edges) between two nodes
   7. Psuedo Graph
      1. Contains self loops (an edge from one node to itself)
2. **Based on Connectivity**
   1. Connected Graph
      1. There is a path between every pair of vertices (e.g. social network where everyone is somehow connected)
   2. Disconnected graph
      1. Some Nodes are isolated (no connection to rest)
3. **Based on Direction and Weight**
   1. Directed Graph (Digraph)
      * Edges have direction(A -> B is not B-> A)
   2. Undirected Graph
      * Edges do not have direction (A- B)
   3. Weighted Graph
      * Each edge has weight or cost (distance, time)
   4. Unweighted Graph
      * All edges are treated equally
4. **Based on cycles**
   1. Cyclic Graph
      * Contains at least one cycle (e.g. A->B->C->A)
   2. Acyclic Graph
      * No cycles present
   3. Directed Acyclic Graph (DAG)
      * Directed and acyclic
      * Scheduling, Orchastrator, Building systems
5. **Others**
   1. Complete Graph
      * Every node is connected to every other node
   2. Regular Graph
      * All vertices have same degree
   3. Sub Graph
      * A subset of the original graph’s vertices and edges

**Graph Representations:**

**Adjacency matrix**

2D matrix – adj[i][j] = 1, if there’s any edge from vertex i to j

Space: O(V2)

A B C

A-B B-C

|  |  |  |  |
| --- | --- | --- | --- |
|  | **A** | **B** | **C** |
| **A** | **0** | **1** | **0** |
| **B** | **1** | **0** | **1** |
| **C** | **0** | **1** | **0** |

**Pros:**

Fast edge lookup : O(1)

**Cons:**

Space inefficient for sparse graphs

**B. Ajdacency List**

- Each vertex stores list of adjacent vertices

A -> B

B ->A, C

C -> B

Pros:

* Efficient for spase graphs
* Easy to integrate neighbors

Cons:

* Slower edge lookup

------------------------------------------------------------------------------------------------------------------------------------------  
**08-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**Assignment**:

For given array return transpose array.

Transpose Array

1 2 3

4 5 6

1. 4
2. 5
3. 6

3 X2 result

Multi-Dimensional Arrays - Two Dimensional Arrays

[2,3,4,5]

[ [1,2], [3,4] , [5,6] ] -2D array

1 2

3 4

5 6

3 X2 array

Int[][] arr = new int[3][3];

Int[][] arr2 = {{1,2}, {3,4}, {5,6}}

1 2 5 6

3 4 7 8

6 8

10 12

**Linked List**

A linked list is a linear data structure where elements are stored in nodes.

Each node has data and a pointer to the next node.

Why Linked List?

* Dynamic size
* Efficient Insertion and Deletion operations

Types of linked lists:

* Singly Linked List - Each node points to the next
* Doubly Linked List - Each node points to the both next and previous nodes
* Circular Linked List - Last node points to the first node

1,2,3

1,2,3,4

|  |  |  |
| --- | --- | --- |
| Feature | Array | LinkedList |
| Size | Fixed (Static) | Dynamic (Grows/s |
| Insert/Delete | Costly (shift elements) | Fast (change pointers) |
| Access time | Fast (random access) | Slower (sequential access) |
| Memory Allocation | Contiguous | Non-Contigous |

**Stack:**

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle.

* You can add (push) and remove (pop) elements from the top of the stack
* Pile of plates 🡪

**Queue:**

* A queue is linear data structure that follow the FIFO (First In, First Out) Principle
  + Elements are added at the rear (enqueue) and removed from the front(dequeue).
  + Think of it as a line at a ticket counter.

------------------------------------------------------------------------------------------------------------------------------------------  
**07-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

Datastructure –

Way of organizing and storing data so that it can be accessed and modified efficiently.

**Commons operations in Data structures:**

* Insertion
* Deletion
* Searching
* Traversal
* Sorting

**Examples:**

* **Array** – Fixed size collection of elements of the same type
* **Linked List** – Collection of nodes connected by references
* **Stacks** – LIFO (Last In First Out) data structure
* **Queues** – FIFO (First in First Out) data structure
* **Trees** – Hierarchical data structure
* **Graphs**: Set of nodes connected by edges

**DataStructures vs Data Types**

**Data Type:** Type of variable

Int, float

**Why need for data structure?**

* Efficient data storage and access
* Reduce Time Complexity of operations
* Better organization for complex data
* Essential in system design and scalable applications

**“**Alice”, 23, BEIT, BOB,24, MCA

[

{name: Alice, age: 23, course: BEIT},

{name: Bob, age: 24, course: MCA}

]

Classification of Data structures:

**Linear Data structures**:

Arrays, Linked Lists, Stack, Queues

[1] 🡪 [2] 🡪 [3] 🡪 [4]

**Non-Linear Data structures:**

Trees, Graphs

1

**Algo Characteristics:**

* Takes input
* Gives Output
* Finiteness
* Effectiveness

**Assignment**:

1. Write a program to find the sum of all elements in a integer array

[2,4,6,8] 🡪 **output**: 20

1. Write program to reverse the elements of an array:

[1,2,3,4] 🡪 **Output**: [4,3,2,1]

1. Find maximum and Minimum in an array

Input: [5,8,3,1,9] 🡪 **Output** : Max: 9, Min: 1

1. Count even and Odd numbers in an array

**Input**: [2,3,5,6,8] 🡪 **Output**: Even : 3, Odd: 2

1. Print all elements at even indexes:

**Input**: [10,20,30,40,50] 🡪 **Output**: 10 30 50

**Time Complexity**

Amount of time taken by algorithm to run.

Time and Space complexity can define effectiveness of an algorithm.

**Big O Notation** – mathematical way to describe the time complexity of an algorithm in terms of input size n. It tells you how the performance of your algorithm scales as the input grows.

Compare different algorithms

**Different types of Time Complexities:**

1. Constant Time – O(1) - Time doesn’t grow with input size
2. Linear Time – O(n) - Grows proportionally with the input size
3. Logarithmic Time : O(log n) - Cuts input in half each time
4. Quadratic Time: O(n^2) - Slower for large input
5. Cubic Time – O(n^3) -
6. Exponential Time - O(2^n) - Very slow, doubles time each step
7. Factorial Time – O(n!) - Extremely slow even for small n

What can cause time in function?

1. Operations (+,-….)
2. Comparisons (>,<,==)
3. Looping (for, while)

Rules:

1. Always consider worst case
2. Remove constants
3. Remove non dominant terms

Iteration: x

1. x
2. x/2
3. x/4
4. x/8

….

K x/2^k

2^k = x

K = log2(x)

**Space Complexity**

Space complexity is the total memory used by an algorithm as a function of input size n.

It includes:

* Memory used by input data
* Memory used by variables
* Memory used by recursion stack or function calls
* Memory used by auxiliary data structures (like arrays, hashmaps exc)