# Question Bank Answers (DSA – IAE1)

CONTENT WARNING: READING THIS DOCUMENT MAY CAUSE SUDDEN BURSTS OF INTELLIGENCE. PROCEED WITH CAUTION.

**1. What is Data Structure?**

A data structure is a way of organizing and storing data in a computer so it can be accessed and

modified efficiently.

**Common types include:**

**Arrays:** A collection of elements of the same type stored in contiguous memory

locations.

**Linked Lists:** A sequence of nodes where each node contains data and a reference to the

next node.

**Stacks:** A LIFO (Last In, First Out) structure used for reversing data.

**Queues:** A FIFO (First In, First Out) structure used in scheduling processes.

**Trees:** A hierarchical structure with nodes connected by edges, often used in databases.

**Graphs:** A set of vertices connected by edges, useful for representing networks.

**2. Differentiate between linear & non-linear data structure**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Linear Data Structures** | **Non-linear Data Structures** |
| **Arrangement** | Elements are arranged in a sequential manner. | Elements are arranged in a hierarchical manner. |
| **Predecessor/Successor** | Each element has a single predecessor and a single successor (except the first and last elements). | An element can have multiple predecessors and successors. |
| **Memory Allocation** | Memory is allocated in a contiguous block. | Memory is allocated in a non-contiguous manner. |
| **Access Time** | Generally allows for faster access time due to direct indexing. | Access time can be slower and not based on direct indexing. |
| **Complexity** | Generally simpler in structure and easier to implement. | More complex due to multiple relationships between elements. |
| **Traversal** | Traversal is straightforward, usually in one pass. | Traversal can be complex and might require specific algorithms. |
| **Examples** | Arrays, Linked Lists, Stacks, Queues. | Trees, Graphs. |

**3. What is stack?**

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle, meaning the last element added is the first one to be removed.

**Primary Operations:**

* **Push:** Adds an element to the top of the stack.
* **Pop:** Removes the element from the top of the stack.

**Example:**

Think of a stack of plates: when you add (push) a plate, it goes on top, and when you remove (pop) a plate, you take the top one first.

**Applications:**

Stacks are commonly used in various applications, including:

* Function call management in programming.
* Undo mechanisms in text editors.
* Expression evaluation in compilers.

**4. What is queue?**

A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle, meaning the first element added is the first one to be removed.

**Main Operations:**

* **Enqueue:** Adds an element to the end of the queue.
* **Dequeue:** Removes the element from the front of the queue.

**Example:**

Imagine a line at a ticket counter: the first person in line is served first, and new people join at the end of the line.

**Applications:**

Queues are widely used in various applications, including:

* Task scheduling.
* Managing print jobs.
* Handling requests in servers.

**5. What are different types of queue**

**Types of Queues**

1. **Simple Queue (or Linear Queue):**
   * Description: Follows the basic FIFO principle.
   * Operations: Elements are added at the rear and removed from the front.
   * Example: A line of people waiting to board a bus.
2. **Circular Queue:**
   * Description: The last position is connected back to the first position, forming a circle.
   * Benefits: Overcomes the limitation of unused space in a simple queue.
   * Example: CPU scheduling in operating systems.
3. **Priority Queue:**
   * Description: Each element is assigned a priority; elements with higher priority are dequeued before those with lower priority.
   * Example: Emergency room patients, where more critical cases are treated first.
4. **Double-Ended Queue (Deque):**
   * Description: Allows insertion and deletion at both the front and rear ends.
   * Example: Undo functionality in text editors.
5. **Input-Restricted Deque:**
   * Description: Insertion is allowed at only one end, but deletion is allowed at both ends.
   * Example: Managing dual-ended task queues where tasks can be removed from either end.
6. **Output-Restricted Deque:**
   * Description: Deletion is allowed at only one end, but insertion is allowed at both ends.
   * Example: A queue where new tasks can be added at either end, but tasks are processed from one end only.

**6. What is array implementation stack**

An array implementation of a stack uses an array to represent the stack data structure.   
Here’s how it works:

**Operations:**

1. **Push (Element):**
   * **Action:** Add an element to the top of the stack.
   * **Process:**
     + Check if the stack is full (i.e., if top == size - 1). If it is full, display an overflow error.
     + If not full, increment top by 1 and insert the element at the top index of the array.
2. **Pop:**
   * **Action:** Remove the top element from the stack.
   * **Process:**
     + Check if the stack is empty (i.e., if top == -1). If it is empty, display an underflow error.
     + If not empty, return the element at the top index and decrement top by 1.

**7. What is array implementation queue**

The array implementation of a queue uses an array to represent the queue data structure. Here’s how it works:

**Operations:**

1. **Enqueue (Element):**
   * **Action:** Add an element to the rear of the queue.
   * **Process:**
     + Check if the queue is full (i.e., if rear == size - 1). If it is full, display an overflow error.
     + If not full, increment rear by 1 and insert the element at the rear index of the array.

* **Dequeue:**
  + **Action:** Remove the front element from the queue.
  + **Process:**
    - Check if the queue is empty (i.e., if front == rear + 1). If it is empty, display an underflow error.
    - If not empty, return the element at the front index and increment front by 1.

**8. What is circular queue**

A circular queue is a type of queue where the last position is connected back to the first position, forming a circle. This design allows for efficient space utilization by reusing positions in the array that have been vacated by dequeued elements.

**Operations:**

1. **Enqueue (Element):**
   * **Action:** Add an element to the rear of the queue.
   * **Process:**
     + Calculate the new position of rear using (rear + 1) % size.
     + Check if the queue is full (i.e., if (rear + 1) % size == front). If it is full, display an overflow error.
     + If not full, insert the element at the calculated rear position.
2. **Dequeue:**
   * **Action:** Remove the front element from the queue.
   * **Process:**
     + Check if the queue is empty (i.e., if front == rear + 1). If it is empty, display an underflow error.
     + If not empty, retrieve the element at the front, then move front to (front + 1) % size.

**9. What is double ended queue**

A double-ended queue (deque) is a linear data structure that allows insertion and deletion of elements at both the front and rear ends. This flexibility makes it more versatile compared to simple queues or stacks.

**Types of Deques:**

1. **Input-Restricted Deque:**
   * **Insertion:** Allowed only at the rear end.
   * **Deletion:** Allowed from both the front and rear ends.
   * **Example:** A queue where new elements are added only at the back, but can be removed from either end.
2. **Output-Restricted Deque:**
   * **Insertion:** Allowed at both the front and rear ends.
   * **Deletion:** Allowed only from the front end.
   * **Example:** A queue where elements can be added at either end, but can only be removed from the front.

**10. What is linked list**

A linked list is a linear data structure where elements, known as nodes, are stored in non-contiguous memory locations. Each node contains data and a reference (or link) to the next node in the sequence. This structure allows for efficient insertion and deletion of elements compared to arrays.

**11. Types of linked list**

1. **Singly Linked List:**
   * Each node points to the next node.
   * **Example:** A list where each node contains a single link to the next node, allowing for traversal in one direction.
2. **Doubly Linked List:**
   * Each node has two links: one to the next node and one to the previous node.
   * **Example:** A list where each node has links to both its next and previous nodes, enabling traversal in both directions.
3. **Circular Linked List:**
   * The last node points back to the first node, forming a circle.
   * **Example:** A list where the end connects back to the start, making it useful for applications like round-robin scheduling.
4. **Doubly Circular Linked List:**
   * Combines features of both doubly linked lists and circular lists.
   * **Example:** A circular list where each node links to both its previous and next nodes, facilitating traversal in both directions while maintaining a circular structure.

**12. How to do insertion, deletion and updation of queue**

In a queue, the basic operations are insertion, deletion, and updating. Here’s how each operation is typically performed using a circular queue implementation with an array:

**1. Insertion (Enqueue)**

* **Action:** Add an element to the rear of the queue.
* **Steps:**
  + Check if the queue is full.
  + If not full, increment the rear index and insert the new element.

**Example Code:**

void enqueue(int queue[], int \*rear, int front, int size, int element) {

if ((\*rear + 1) % size == front) { // Check for overflow

printf("Queue Overflow\n");

} else {

\*rear = (\*rear + 1) % size; // Increment rear index circularly

queue[\*rear] = element;

}

}

**2. Deletion (Dequeue)**

* **Action:** Remove an element from the front of the queue.
* **Steps:**
  + Check if the queue is empty.
  + If not empty, retrieve the element from the front and increment the front index.

**Example Code:**

int dequeue(int queue[], int \*front, int rear, int size) {

if (\*front == (rear + 1) % size) { // Check for underflow

printf("Queue Underflow\n");

return -1; // Return a sentinel value or handle underflow

} else {

int element = queue[\*front];

\*front = (\*front + 1) % size; // Increment front index circularly

return element;

}

}

**3. Update**

* **Action:** Update the value of an element at a specific position.
* **Steps:**
  + Calculate the position based on front and rear indices.
  + Access and modify the element at that position.

Example Code:

void update(int queue[], int front, int rear, int size, int position, int newValue) {

int count = (rear + size - front + 1) % size; // Number of elements in the queue

if (position < 0 || position >= count) {

printf("Invalid Position\n");

} else {

int actualIndex = (front + position) % size;

queue[actualIndex] = newValue;

}

}

**Explanation:**

* Insertion: Adds an element at the rear end of the queue.
* Deletion: Removes an element from the front end of the queue.
* Update: Modifies the value of an element at a specified position.

These operations are implemented in a circular queue to efficiently utilize space and handle wrap-around. The provided code assumes the use of a circular array-based queue.

**13. What is tree and explain its terminology**

A tree is a hierarchical data structure consisting of nodes, where each node contains data and references (or links) to other nodes. Trees are used to represent hierarchical relationships and are widely applied in various domains such as databases, file systems, and more.

**Terminology:**

1. **Node:**
   * Definition: A fundamental part of a tree containing data and links to other nodes.
   * Example: Each node in a family tree represents a person.
2. **Root:**
   * Definition: The topmost node of the tree, serving as the starting point.
   * Example: In a directory structure, the root directory is the topmost directory.
3. **Parent:**
   * Definition: A node that has one or more child nodes.
   * Example: In a family tree, a parent node represents a person who has one or more children.
4. **Child:**
   * Definition: A node that is a descendant of another node (its parent).
   * Example: In a file system, files inside a folder are child nodes of that folder.
5. **Leaf (or External Node):**
   * Definition: A node with no children, indicating the end of a branch.
   * Example: In a family tree, a leaf node could represent a person with no descendants.
6. **Subtree**:
   * Definition: A tree formed by a node and its descendants.
   * Example: In a directory structure, a subtree includes a folder and all its contents.
7. **Level:**
   * Definition: The level of a node is its distance from the root node, with the root at level 0.
   * Example: In an organizational chart, employees at different hierarchical levels have different levels.
8. **Height:**
   * Definition: The length of the longest path from a node to a leaf. The height of a tree is determined by its root node.
   * Example: The height of a directory structure is the longest path from the root directory to the deepest file or folder.
9. **Depth:**
   * Definition: The length of the path from a node to the root node.
   * Example: In a family tree, the depth of a node indicates the number of generations between it and the root ancestor.
10. **Degree:**
    * Definition: The number of children a node has.
    * Example: In a family tree, the degree of a person (node) is the number of their children.
11. **Binary Tree:**
    * Definition: A type of tree where each node has at most two children, referred to as the left child and the right child.
    * Example: A binary search tree used in database indexing.
12. **Binary Search Tree (BST):**
    * Definition: A binary tree where, for each node, all elements in the left subtree are smaller, and all elements in the right subtree are larger.
    * Example: Used for efficient searching and sorting operations.

**Example Diagram:**

Root

/ \

Node1 Node2

/ \ \

Node3 Node4 Node5

* **Root:** The top node.
* **Node1 and Node2:** Children of the root.
* **Node3 and Node4:** Children of Node1.
* **Node5:** A child of Node2.

**14. Types of binary tree**

1. **Full Binary Tree**:
   * **Definition**: Every node has either 0 or 2 children.
   * **Properties**: All non-leaf nodes have exactly two children.
   * **Example**:



1. **Complete Binary Tree**:
   * **Definition**: All levels are fully filled except possibly for the last level, which is filled from left to right.
   * **Properties**: The last level has all nodes as far left as possible.
   * **Example**:



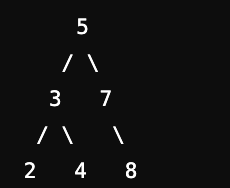
1. **Perfect Binary Tree**:
   * **Definition**: All internal nodes have exactly two children, and all leaf nodes are at the same level.
   * **Properties**: It is both full and complete.
   * **Example**:



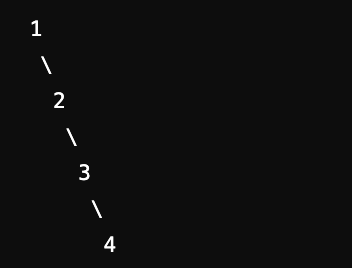
1. **Balanced Binary Tree**:
   * **Definition**: The height of the left and right subtrees of every node differ by no more than one.
   * **Properties**: Ensures O(log n) height, leading to efficient operations.
   * **Example**: AVL Tree (a type of self-balancing binary search tree).



1. **Binary Search Tree (BST)**:
   * **Definition**: A binary tree where for each node, all nodes in the left subtree have values less than the node's value, and all nodes in the right subtree have values greater.
   * **Properties**: Facilitates efficient searching, insertion, and deletion.
   * **Example**:



1. **Degenerate (or Pathological) Tree**:
   * **Definition**: A binary tree where each parent node has only one child, creating a structure similar to a linked list.
   * **Properties**: This structure has O(n) height, leading to inefficient operations.
   * **Example**:



1. **Trie (Prefix Tree)**:
   * **Definition**: A special kind of tree used to store associative data structures. Each node represents a common prefix of some strings.
   * **Properties**: Often used for storing dictionaries or implementing autocomplete features.
   * **Example**:



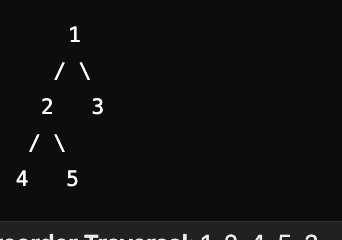
* + **Strings**: "abc", "abd", "be"

Each type of binary tree has specific applications and is chosen based on requirements for efficiency, balancing, and structural characteristics.

**15. Explain pre-order, in-order and post-order**

Traversal of a binary tree involves visiting each node in a specific order. The three main types of depth-first traversal are preorder, inorder, and postorder. Each method follows a unique sequence:

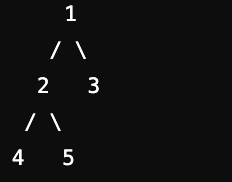
**1. Preorder Traversal**

* **Definition**: Visit the root node first, then recursively traverse the left subtree, followed by the right subtree.
* **Order**: Root → Left → Right
* **Algorithm**:
  1. Visit the root node.
  2. Traverse the left subtree.
  3. Traverse the right subtree.
* **Example**: For the tree:

Preorder Traversal: 1, 2, 4, 5, 3

**2. Inorder Traversal**

* **Definition**: Recursively traverse the left subtree first, visit the root node, then recursively traverse the right subtree.
* **Order**: Left → Root → Right
* **Algorithm**:
  1. Traverse the left subtree.
  2. Visit the root node.
  3. Traverse the right subtree.
* **Example**: For the same tree:



**Inorder Traversal**: 4, 2, 5, 1, 3

**3. Postorder Traversal**

* **Definition**: Recursively traverse the left subtree, then the right subtree, and finally visit the root node.
* **Order**: Left → Right → Root
* **Algorithm**:
  1. Traverse the left subtree.
  2. Traverse the right subtree.
  3. Visit the root node.
* **Example**: For the same tree:



**Postorder Traversal**: 4, 5, 2, 3, 1

These traversal methods are fundamental for various applications, including expression tree evaluations and syntax tree traversals in compilers.