**DDS Question Bank Answers**

**### 1. What is a Data Structure? Explain its Importance.**

**A \*\*data structure\*\* is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently. Examples include arrays, linked lists, stacks, queues, trees, and graphs. The importance of data structures lies in their ability to manage and organize data in a way that optimizes resource usage and performance. Efficient data structures are critical for writing optimized algorithms, improving program execution time, and managing large data sets.**

**### 2. Differentiate Between Linear and Non-Linear Data Structures.**

**- \*\*Linear Data Structures:\*\* Elements are arranged in a sequential order. Examples include arrays, linked lists, stacks, and queues. In linear structures, elements have a single predecessor and successor (except the first and last elements).**

**- \*\*Non-Linear Data Structures:\*\* Elements are arranged in a hierarchical or interconnected manner. Examples include trees and graphs. In non-linear structures, elements can have multiple predecessors and successors, allowing for more complex relationships.**

**### 3. What are the Primary Operations that can be Performed on a Stack?**

**The primary operations on a stack are:**

**- \*\*Push:\*\* Adds an element to the top of the stack.**

**- \*\*Pop:\*\* Removes the element from the top of the stack.**

**- \*\*Peek/Top:\*\* Returns the top element without removing it.**

**- \*\*IsEmpty:\*\* Checks whether the stack is empty.**

**- \*\*IsFull:\*\* Checks whether the stack is full (applicable in fixed-size stacks).**

**### 4. Describe the Difference Between a Stack and a Queue.**

**- \*\*Stack:\*\* Follows a Last-In-First-Out (LIFO) approach, meaning the last element added is the first one to be removed.**

**- \*\*Queue:\*\* Follows a First-In-First-Out (FIFO) approach, meaning the first element added is the first one to be removed.**

**### 5. How does a Circular Queue Differ from a Regular Queue?**

**In a \*\*regular queue\*\*, once the rear pointer reaches the end of the queue, no more elements can be added, even if there is space at the front due to dequeued elements. A \*\*circular queue\*\* solves this problem by connecting the end of the queue back to the front, allowing the queue to reuse the space and function efficiently in a circular manner.**

**### 6. Explain the Concept of a Linked List. How is it Different from an Array?**

**A \*\*linked list\*\* is a linear data structure where elements (called nodes) are linked together using pointers. Each node contains data and a reference to the next node in the sequence.**

**- \*\*Difference from an Array:\*\***

**- \*\*Memory Allocation:\*\* Arrays have a fixed size and are stored in contiguous memory locations, while linked lists are dynamic and can grow or shrink as needed, with nodes stored at non-contiguous memory locations.**

**- \*\*Access Time:\*\* Arrays provide O(1) access time for elements via indices, whereas linked lists require O(n) time to access an element.**

**### 7. Difference Between Array and Linked List**

**- \*\*Size:\*\* Arrays have a fixed size, while linked lists are dynamic.**

**- \*\*Memory:\*\* Arrays require contiguous memory allocation, while linked lists do not.**

**- \*\*Access:\*\* Arrays offer fast, direct access via indices, whereas linked lists require traversal.**

**- \*\*Insertion/Deletion:\*\* Linked lists allow easier and more efficient insertion and deletion of elements, especially in the middle of the structure, while arrays require shifting elements.**

**### 8. What are the Main Operations of a Queue? Describe Their Time Complexities.**

**The main operations on a queue are:**

**- \*\*Enqueue:\*\* Adding an element to the rear of the queue (O(1)).**

**- \*\*Dequeue:\*\* Removing an element from the front of the queue (O(1)).**

**- \*\*Peek/Front:\*\* Accessing the front element without removing it (O(1)).**

**- \*\*IsEmpty:\*\* Checking whether the queue is empty (O(1)).**

**### 9. What are the Main Operations of a Stack?**

**The main operations on a stack include:**

**- \*\*Push:\*\* Inserting an element at the top of the stack (O(1)).**

**- \*\*Pop:\*\* Removing the element from the top of the stack (O(1)).**

**- \*\*Peek:\*\* Viewing the top element without removing it (O(1)).**

**- \*\*IsEmpty:\*\* Checking if the stack is empty (O(1)).**

**### 10. Describe the Algorithm for Evaluating Postfix Expressions Using a Stack.**

**1. \*\*Initialize\*\* an empty stack.**

**2. \*\*Traverse\*\* the postfix expression from left to right.**

**3. \*\*For each token:\*\***

**- If the token is an operand, \*\*push\*\* it onto the stack.**

**- If the token is an operator, \*\*pop\*\* the required number of operands from the stack, \*\*apply\*\* the operator, and \*\*push\*\* the result back onto the stack.**

**4. After the expression is fully traversed, the \*\*final result\*\* will be the only element left in the stack.**

**### 11. Explain the Concept of a Singly Linked List and Provide an Example.**

**A \*\*singly linked list\*\* is a type of linked list where each node points to the next node in the sequence, and the last node points to `null`, indicating the end of the list. Example:**

**```**

**Head -> [Data: 1, Next: Node2] -> [Data: 2, Next: Node3] -> [Data: 3, Next: null]**

**```**

**### 12. How does a Doubly Linked List Differ from a Singly Linked List?**

**A \*\*doubly linked list\*\* has nodes that contain references to both the next and previous nodes, allowing traversal in both directions. In contrast, a singly linked list only allows traversal in one direction (forward) as each node only points to the next node.**

**### 13. What is a Circular Linked List, and How is it Implemented?**

**A \*\*circular linked list\*\* is a type of linked list where the last node’s next pointer points back to the first node, forming a circle. It can be implemented by modifying the last node's next pointer to reference the head of the list.**

**### 14. What is Linear Search, and What is its Time Complexity?**

**\*\*Linear search\*\* is a simple search algorithm that checks each element in a list sequentially until the desired element is found or the list is exhausted. The time complexity is \*\*O(n)\*\*, where `n` is the number of elements in the list.**

**### 15. Explain Binary Search. How Does It Improve Search Efficiency Compared to Linear Search?**

**\*\*Binary search\*\* is an efficient algorithm that finds the position of a target value within a sorted array. It repeatedly divides the search interval in half, comparing the target value to the middle element. If they are not equal, the half in which the target cannot lie is eliminated. The time complexity is \*\*O(log n)\*\*, which is more efficient than linear search’s O(n) for large datasets.**

**### 16. How Does Selection Sort Work?**

**\*\*Selection sort\*\* is a comparison-based sorting algorithm that divides the input list into a sorted and an unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part. The time complexity is \*\*O(n²)\*\*.**

**### 17. Classify Data Structures with Diagram.**

**\*\*Classification of Data Structures:\*\***

**1. \*\*Linear Data Structures:\*\***

**- Arrays**

**- Linked Lists**

**- Stacks**

**- Queues**

**2. \*\*Non-Linear Data Structures:\*\***

**- Trees**

**- Graphs**

**\*(A diagram can visually represent the above classification, where linear structures are organized sequentially and non-linear structures are branched.)\***

**### 18. Interpret Big O Complexity Chart.**

**\*\*Big O Notation\*\* describes the upper limit of the time complexity of an algorithm in terms of the input size. Common Big O complexities are:**

**- \*\*O(1):\*\* Constant time, independent of input size.**

**- \*\*O(log n):\*\* Logarithmic time, reduces problem size by a constant factor.**

**- \*\*O(n):\*\* Linear time, directly proportional to input size.**

**- \*\*O(n log n):\*\* Linearithmic time, common in efficient sorting algorithms.**

**- \*\*O(n²):\*\* Quadratic time, grows quadratically with input size.**

**- \*\*O(2^n):\*\* Exponential time, doubles with each additional element.**

**- \*\*O(n!):\*\* Factorial time, extremely inefficient for large inputs.**

**### 19. Discuss Time Complexity.**

**\*\*Time complexity\*\* is a computational concept that describes the amount of time an algorithm takes to complete as a function of the length of the input. It helps in determining the efficiency of an algorithm and predicting how it will scale as the Input size increases.**

**### 20. Describe Sparse Matrix. Find the Address of A[2][1] if Base Address is 1024 for an Integer Array A[5][4] in Row-Major Order and Word Size is 2 Bytes.**

**A \*\*sparse matrix\*\* is a matrix in which most elements are zero. It is often used to save space and computational resources.**

**\*\*Finding the Address:\*\***

**- \*\*Given:\*\* Base address = 1024, word size = 2 bytes, array A[5][4].**

**- \*\*Row-major order:\*\* Address of A[i][j] = Base Address + [(I × Number of columns) + j] × Element size.**

**- \*\*Substitute values:\*\* Address of A[2][1] = 1024 + [(2 × 4) + 1] × 2 = 1024 + [8 + 1] × 2 = 1024 + 18 = \*\*1042\*\*.**

**### 22. Define Dynamic Memory Allocation.**

**\*\*Dynamic memory allocation\*\* is the process of allocating memory during the runtime of a program. It allows programs to request memory as needed using functions like `malloc`, `calloc`, `realloc`, and `free` in C, or the `new` and `delete` operators in C++. This flexibility is essential for managing memory efficiently in applications where the amount of data is not known at compile time.**

**### 23. Define Referential Structure.**

**A \*\*referential structure\*\* refers to a data structure where elements contain references (or pointers) to other elements rather than containing the actual data themselves. Examples include linked lists, trees, and graphs where nodes reference other nodes. These structures are used to model complex relationships and allow for dynamic memory allocation and flexible memory usage.**

**### 24. Array is a Heterogeneous Data Type. (True/False). Justify Your Answer.**

**\*\*False.\*\* An array is a homogeneous data type, meaning that all elements in an array are of the same type. For example, an array of integers can only contain integers, and an array of strings can only contain strings. This uniformity allows for efficient memory allocation and access.**

**### 25. A \( m \times n \) Matrix Which Contains Very Few Non-Zero Elements. A Matrix Contains More Number of ZERO Values than NON-ZERO Values. Such Matrix is Known as?**

**Such a matrix is known as a \*\*Sparse Matrix\*\*. In a sparse matrix, most of the elements are zero, and only a few elements are non-zero. This type of matrix is typically used in applications where memory efficiency is critical, and specialized data structures like linked lists, arrays, or dictionaries are used to store only the non-zero elements.**

**### 29. Differentiate Between LIFO and FIFO Access Mechanism.**

**- \*\*LIFO (Last In, First Out):\*\* The last element added is the first to be removed. Used in stacks.**

**- \*\*FIFO (First In, First Out):\*\* The first element added is the first to be removed. Used in queues.**

**### 30. How is Linked List Better Compared to Stack, Queue, and Array? Explain with Concept of Dynamic Memory Allocation.**

**- \*\*Dynamic Memory Allocation:\*\* Linked lists allow dynamic memory allocation, meaning they can grow and shrink in size during runtime, unlike arrays that have a fixed size.**

**- \*\*Better Flexibility:\*\* Linked lists do not require contiguous memory, making them better suited for applications where memory allocation is unpredictable or fragmented.**

**- \*\*Ease of Insertion/Deletion:\*\* Linked lists offer more efficient insertion and deletion operations compared to arrays, especially for large data sets, as no shifting of elements is required.**

**### 31. In which type of scenario is a linear queue (simple queue) better than a circular queue?**

**A linear queue is better in scenarios where the order of processing follows a strict “First In, First Out” (FIFO) principle and the total number of elements is well within the limits of the queue size. For example, when dealing with a sequence of tasks that must be completed in the exact order they are received, with no concern for efficiently reusing memory (i.e., the queue doesn’t need to wrap around to the beginning).**

**### 32. After evaluation of `3, 5, 4, \*, +`, what is the result?**

**To evaluate the postfix expression `3, 5, 4, \*, +`, follow these steps:**

**1. Push 3 onto the stack.**

**2. Push 5 onto the stack.**

**3. Push 4 onto the stack.**

**4. Pop the top two elements (5 and 4) and multiply: `5 \* 4 = 20`. Push 20 onto the stack.**

**5. Pop the top two elements (3 and 20) and add: `3 + 20 = 23`. Push 23 onto the stack.**

**Final result = \*\*23\*\*.**

**### 33. What will be the value of Front and Rear pointers when Queue is empty?**

**When a queue is empty:**

**- The \*\*Front\*\* pointer is usually set to `-1` (or sometimes `0` depending on implementation).**

**- The \*\*Rear\*\* pointer is also set to `-1`.**

**This indicates that there are no elements in the queue.**

**### 34. Apply selection sort algorithm on the following input: `12, 29, 25, 8, 32, 17, 40`. Explain step by step.**

**Selection Sort steps:**

* **\*\*Initial List\*\*: `12, 29, 25, 8, 32, 17, 40`**

**1. \*\*Step 1\*\*: Find the minimum value in the list (8). Swap it with the first element.**

**- \*\*After 1st pass\*\*: `8, 29, 25, 12, 32, 17, 40`**

**2. \*\*Step 2\*\*: Find the next minimum value (12). Swap it with the second element.**

**- \*\*After 2nd pass\*\*: `8, 12, 25, 29, 32, 17, 40`**

**3. \*\*Step 3\*\*: Find the next minimum value (17). Swap it with the third element.**

**- \*\*After 3rd pass\*\*: `8, 12, 17, 29, 32, 25, 40`**

**4. \*\*Step 4\*\*: Find the next minimum value (25). Swap it with the fourth element.**

**- \*\*After 4th pass\*\*: `8, 12, 17, 25, 32, 29, 40`**

**5. \*\*Step 5\*\*: Find the next minimum value (29). Swap it with the fifth element.**

**- \*\*After 5th pass\*\*: `8, 12, 17, 25, 29, 32, 40`**

**6. \*\*Step 6\*\*: The last element is already in place. No need to swap.**

**- \*\*Final Sorted List\*\*: `8, 12, 17, 25, 29, 32, 40`**

**### 35. Write an algorithm for bubble sort. Apply it on random 8 input data.**

**\*\*Bubble Sort Algorithm:\*\***

**1. Start from the first element in the array.**

**2. Compare the current element with the next element.**

**3. If the current element is greater than the next element, swap them.**

**4. Move to the next element and repeat steps 2 and 3 until the end of the array is reached.**

**5. Repeat the entire process for all elements except the last sorted elements.**

**6. Continue until no swaps are needed, indicating the array is sorted.**

**\*\*Applying Bubble Sort on random 8 input data (`23, 15, 7, 9, 1, 31, 11, 19`):\*\***

1. **\*\*Pass 1\*\*: Compare and swap adjacent elements if needed:**

**`15, 7, 9, 1, 23, 11, 19, 31`**

1. **\*\*Pass 2\*\*: Continue swapping:**

**`7, 9, 1, 15, 11, 19, 23, 31`**

1. **\*\*Pass 3\*\*: Continue swapping:**

**`7, 1, 9, 11, 15, 19, 23, 31`**

1. **\*\*Pass 4\*\*: Continue swapping:**

**`1, 7, 9, 11, 15, 19, 23, 31`**

1. **\*\*Pass 5\*\*: No swaps needed, the array is sorted.**

**\*\*Final Sorted List\*\*: `1, 7, 9, 11, 15, 19, 23, 31`**

**###36. Write Merge Sort algorithm. Apply the algorithm to the following elements: 10,5,28, 7, 39, 310, 55,15,1**

**Merge Sort Algorithm**

**Merge Sort is a divide-and-conquer algorithm that works by recursively splitting an array into smaller subarrays until each subarray contains a single element, then merging those subarrays in a sorted manner. Here’s the algorithm in pseudocode:**

**1. \*\*MergeSort(array)\*\***

**- If the array has one or zero elements, return it.**

**- Divide the array into two halves.**

**- Recursively apply MergeSort to both halves.**

**- Merge the sorted halves into a single sorted array.**

**2. \*\*Merge(left, right)\*\***

**- Create an empty array to hold the merged result.**

**- Compare the elements from both halves and insert the smaller element into the result array.**

**- If one half is exhausted, append the remaining elements from the other half to the result.**

**- Return the merged array.**

**### Applying Merge Sort to the Elements: 10, 5, 28, 7, 39, 310, 55, 15, 1**

**Let’s walk through the merge sort process for this array:**

**\*\*Original Array\*\*: [10, 5, 28, 7, 39, 310, 55, 15, 1]**

**1. \*\*Divide\*\*:**

**- [10, 5, 28, 7, 39] and [310, 55, 15, 1]**

**2. \*\*Further Divide\*\*:**

**- [10, 5, 28] and [7, 39]**

**- [310, 55] and [15, 1]**

**3. \*\*Further Divide\*\*:**

**- [10] and [5, 28]**

**- [7] and [39]**

**- [310] and [55]**

**- [15] and [1]**

**4. \*\*Sort and Merge\*\*:**

**- Merge [5] and [28]: [5, 28]**

**- Merge [10] and [5, 28]: [5, 10, 28]**

**- Merge [7] and [39]: [7, 39]**

**- Merge [310] and [55]: [55, 310]**

**- Merge [15] and [1]: [1, 15]**

**5. \*\*Merge Final Halves\*\*:**

**- Merge [5, 10, 28] and [7, 39]: [5, 7, 10, 28, 39]**

**- Merge [55, 310] and [1, 15]: [1, 15, 55, 310]**

**6. \*\*Final Merge\*\*:**

**- Merge [5, 7, 10, 28, 39] and [1, 15, 55, 310]:**

**- Start with [1] (smallest element) and merge the rest:**

**- [1, 5, 7, 10, 15, 28, 39, 55, 310]**

**\*\*Sorted Array\*\*: [1, 5, 7, 10, 15, 28, 39, 55, 310]**