1. What is a Data Structure? Explain different categories in which data structures can be divided. Give an example for each one.

A **data structure** is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. It defines the relationship between data elements and the operations that can be performed on them.

Categories of Data Structures:

1. **Primitive Data Structures:**
    - These are the basic data types provided by a programming language.
    - Example: Integer, Float, Character, Boolean.
2. **Non-Primitive Data Structures:**
    - These are more complex structures derived from primitive data types.
    - Subcategories:
        - **Linear Data Structures:**
            - Elements are arranged sequentially.
            - Example: Array, Linked List, Stack, Queue.
        - **Non-Linear Data Structures:**
            - Elements are arranged hierarchically.
            - Example: Tree, Graph.
        - **Dynamic Data Structures:**
            - Size can grow or shrink at runtime.
            - Example: Linked List, Dynamic Array.

---

2. List and explain different operations that can be performed on a data structure.

Common operations on data structures include:

1. **Traversal:** Accessing each element of the data structure.
2. **Insertion:** Adding a new element to the data structure.
3. **Deletion:** Removing an element from the data structure.
4. **Searching:** Finding the location of a particular element.
5. **Sorting:** Arranging the elements in a specific order (ascending/descending).
6. **Updating:** Modifying an existing element.

---

3. Define different asymptotic notations used to measure the complexity of an algorithm.

Asymptotic notations describe the behavior of an algorithm in terms of its time or space requirements as the input size grows.

1. **Big-O (O):**

- Represents the upper bound of an algorithm.
- Describes the worst-case scenario.
- Example: O(n2)O(n^2)O(n2) for a nested loop.

2. **Omega (Ω):**
    - Represents the lower bound of an algorithm.
    - Describes the best-case scenario.
    - Example: Ω(n)Ω(n)Ω(n) for a linear search.
3. **Theta (Θ):**
    - Represents the average-case complexity.
    - Describes the tight bound (both upper and lower).
    - Example: Θ(nlogn)Θ(n \log n)Θ(nlogn) for merge sort.
4. **Little-o (o):**
    - Indicates that a function grows strictly slower than another function.
    - Example: o(n2)o(n^2)o(n2) implies growth rate slower than n2n^2n2.
5. **Little-omega (ω):**
    - Indicates that a function grows strictly faster than another function.
    - Example: ω(n)ω(n)ω(n) implies growth rate faster than nnn.

---

4. What is an algorithm? What are the characteristics of an algorithm?

An **algorithm** is a step-by-step procedure or formula for solving a problem or performing a task.

Characteristics of an Algorithm:

1. **Finite:** The algorithm must terminate after a finite number of steps.
2. **Definiteness:** Each step must be clear and unambiguous.
3. **Input:** It should accept zero or more inputs.
4. **Output:** It should produce at least one output.
5. **Effectiveness:** Each step should be simple enough to be performed in a finite amount of time.
6. **Generality:** The algorithm should solve a broad class of problems.

---

5. What is meant by complexity of an algorithm? Explain different types of complexities.

Complexity of an algorithm refers to the resources it requires for execution, primarily time and space.

Types of Complexities:

1. **Time Complexity:**
    - The amount of time an algorithm takes to complete as a function of the input size.
    - Example: O(n)O(n)O(n), O(n2)O(n^2)O(n2).

2. **Space Complexity:**
   - The amount of memory required by an algorithm during execution.
   - Includes space for input, output, and auxiliary storage.
   - Example: O(1)O(1)O(1) for constant extra space, O(n)O(n)O(n) for storing nnn elements.

---

6. Write an algorithm to insert an element into the array and to delete an element from the array.

Algorithm to Insert an Element into an Array:

1. Start.
2. Declare the array `arr[]`, its size `n`, and the position `pos` where the element `x` is to be inserted.
3. Shift all elements from position `pos` to the right.
4. Insert the element `x` at position `pos`.
5. Increment the size `n` of the array.
6. End.

Algorithm to Delete an Element from an Array:

1. Start.
2. Declare the array `arr[]`, its size `n`, and the position `pos` of the element to be deleted.
3. Shift all elements from position `pos + 1` to the left.
4. Decrement the size `n` of the array.
5. End.

Example Code (Insert and Delete in C++):

```cpp
void insertElement(int arr[], int &n, int x, int pos) {
    for (int i = n; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = x;
    n++;
}

void deleteElement(int arr[], int &n, int pos) {
    for (int i = pos; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    n--;
}
```