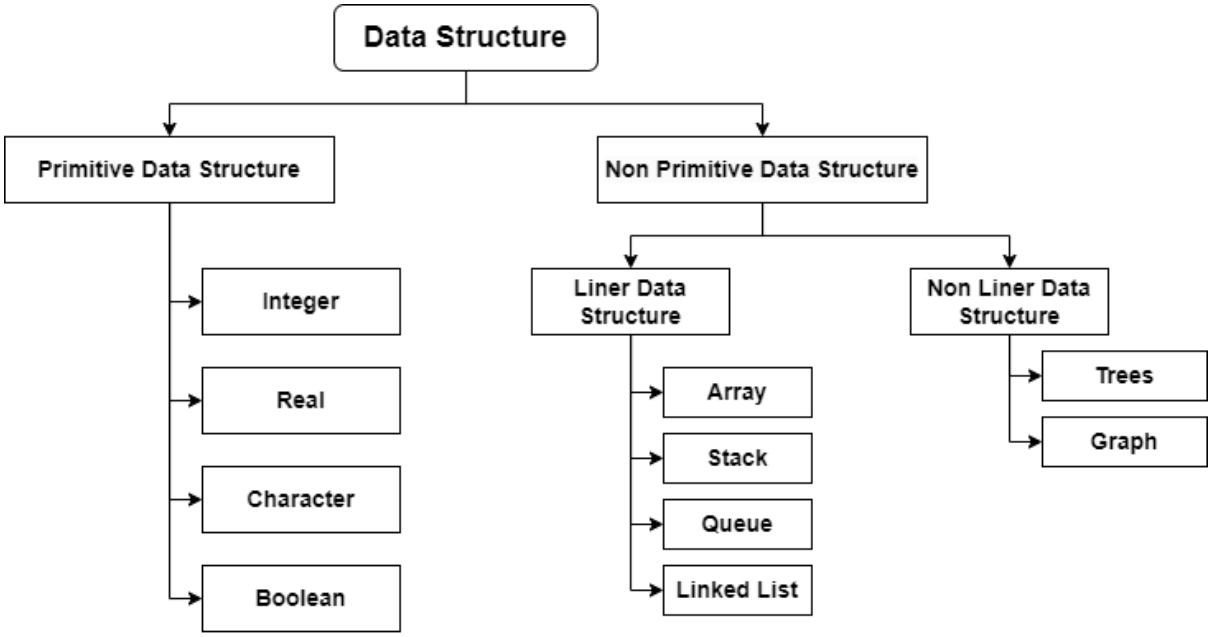
~ Created by Dhruv Prajapati

Q.1 Define Data Structure. Classify the types of data structures.

=> A data structure is a systematic way of organizing and storing data in main memory to facilitate efficient access and modification.



Q2. Explain primitive and non-primitive data structures in detail

=> **Primitive data structures** are the most basic types of data that are directly supported by the system's hardware or the programming language. These data structures operate at the machine level and are often used to build more complex structures.

=> **Integers**: Represents whole numbers, both positive and negative. For example, int in C or Java.

=> **Floats (Floating-point numbers)**: Represents real numbers with fractional parts. For example, float or double in C or Java.

=> **Characters**: Represents single characters or symbols, like char in C or Java.

=> **Booleans**: Represents true or false values, such as bool in C++ or boolean in Java.

=> **Non-primitive data structures** are more complex data structures derived from primitive data structures. They are designed to handle large amounts of data and can be classified into two main categories: linear and non-linear data structures.

=> **Linear Data Structures**

Linear data structures store data in a sequential manner, where each element is connected to its previous and next element.

- **Arrays**:
  - A collection of elements of the same type stored in contiguous memory locations.
  - Example: `int arr[10];` in C.
- **Linked Lists**:
  - A sequence of nodes where each node contains data and a reference to the next node in the sequence.
  - Types: Singly Linked List, Doubly Linked List, Circular Linked List.
- **Stacks**:
  - A collection of elements that follows the Last In, First Out (LIFO) principle.
  - Example: Pushing and popping elements in a function call stack.
- **Queues**:
  - A collection of elements that follows the First In, First Out (FIFO) principle.
  - Types: Simple Queue, Circular Queue, Priority Queue, Deque (Double-ended queue).

=> **Non-Linear Data Structures**

Non-linear data structures store data in a hierarchical manner and are more complex than linear data structures.

- **Trees**:
  - A hierarchical structure with nodes connected by edges, with a single root node.
  - Types: Binary Tree, Binary Search Tree (BST), AVL Tree, Red-Black Tree, Heap, B-Tree, Trie.
- **Graphs**:
  - A collection of nodes (vertices) connected by edges, which can be directed or undirected.
  - Types: Directed Graph, Undirected Graph, Weighted Graph, Unweighted Graph, Cyclic Graph, Acyclic Graph.

Q3. Define Algorithm. Explain key features of an algorithm.

=> Algorithm is a Step by Step Procedure to solving a problem in finite numbers of steps.

*Key Features Of a Algorithm

Finite:

- An algorithm must have a finite number of steps. It should not run indefinitely and must terminate after a certain number of steps, ensuring it eventually produces an output.

Definite:

- Each step of an algorithm must be clear and unambiguous. The instructions should be precisely defined so that there is no confusion about what needs to be done at each step.

Input:

- An algorithm should have zero or more inputs. These are the values or data provided to the algorithm at the beginning, which it will process to produce the output.

Output:

- An algorithm should have one or more outputs. These are the results or the solutions produced after processing the input through the defined steps of the algorithm.

Effectiveness:

- The steps of an algorithm must be basic enough to be performed exactly and in a finite amount of time by a person using paper and pencil or by a computer. Each operation should be simple and feasible.

Generality:

- An algorithm should be general enough to solve a class of problems, not just a single specific problem. It should be applicable to a variety of inputs and produce correct results for all valid inputs.

Q4. Define following.

1. Best Case 2. Worst Case 3. Time Complexity 4. Space Complexity 5. Big O notation 6. Big Omega notation.

=> 1) **Best Case:** The **best case** for an algorithm is the scenario where it performs the minimum number of steps or operations to complete its task. This usually happens with the most favorable input conditions, resulting in the fastest possible execution time.

=> 2) **Worst Case:** The **worst case** for an algorithm is the scenario where it performs the maximum number of steps or operations to complete its task. This happens under the most unfavorable input conditions, leading to the slowest possible execution time. Worst-case analysis is crucial for understanding the upper bounds of an algorithm's performance.

=> 3) **Time Complexity:** Time complexity of an algorithm measures the amount of time it takes to complete as a function of the length of the input. It is a way to express the performance and efficiency of an algorithm, usually in terms of the size of the input (n). Time complexity helps to compare different algorithms and understand their scalability.

=> 4) **Space Complexity**: Space complexity of an algorithm measures the amount of memory space it requires to complete as a function of the length of the input. It includes both the space needed for the input data and the space needed for any additional variables or data structures used by the algorithm. Space complexity is crucial for understanding the memory requirements of an algorithm.

=> 5) **Big O Notation**: **Big O notation** is used to describe the upper bound of an algorithm's time or space complexity. It provides an asymptotic analysis, focusing on the worst-case scenario. Big O notation simplifies complexity expressions by ignoring constant factors and lower-order terms, allowing us to focus on the most significant factors that affect performance.

=> 6) **Big Omega Notation: Big Omega notation (Ω)** is used to describe the lower bound of an algorithm's time or space complexity. It provides an asymptotic analysis, focusing on the best-case scenario. Big Omega notation also simplifies complexity expressions by ignoring constant factors and lower-order terms.

Q5. Write and explain algorithm of linear search and binary search with example.

=> **1. Liner Search**

Linear search is a simple searching algorithm that checks each element of a list sequentially until the desired element is found or the list is exhausted.

```
Algorithm Linersearch(array, size, target) {
        Step 1:  [Initialize]  I  <= 0
        Step 2: Repeat steps 3,4 while  i<size
        Step 3: if(array[i] == target)
                Return i
        Step 4: I <= I + 1
        Step 5: return -1
        Step 6: exit
}
```

- Best case time complexity is order of bigO’1’
- Average case bigO(n)
- Worst case bigO(n)

=> **2. Binary Search**

Binary search is a more efficient searching algorithm that works on sorted lists. It repeatedly divides the search interval in half and compares the target value with the middle element of the current interval. If the middle element is not the target, it eliminates half of the interval and continues the search in the remaining half.

- Time Complexity: O(log⁡n)O(\log n)O(logn)
- Requires the list to be sorted.
- More efficient for large datasets due to its logarithmic time complexity

```
Algorithm Binarysearch(array, size, target) {
        Step 1:  low  <= 0
        Step 2:  high <= size-1
        Step 3: Repeat Steps 4, 5  while  low<=high
        Step 4: mid <= (low+high) / 2
        Step 5: if (array[mid] > target){
                        high = mid-1
                } else if (array[mid] < target) {
                        low = mid+1
                }
                else {
                        return mid
                }
        Step 6: return -1
        Step 7: exit
}
```

Q6. Discuss row major arrays and column major arrays.

=> **1. Row Major arrays**

=> In row major computer will store elements of 2D array row wise in consecutive memory location.

=> In **column-major order**, the elements of a multi-dimensional array are stored column by column. This means all elements of the first column are stored in contiguous memory locations, followed by all elements of the second column, and so on.

1 2 3

4 5 6

computer Will store like this

0,0	0,1	0,2	1,0	1,1	1,2
1	2	3	4	5	6
300	304	308	312	316	320

=> address[array[I][j]] = baseaddress + w (n (i-li) + (j-lj))

N = number of column I = row index j = column index

li = starting index of row lj = starting index of column

=> **2. Column major arrays**

=> In column major computer will store elements of 2D array column wise in consecutive memory location.

=> In **column-major order**, the elements of a multi-dimensional array are stored column by column. This means all elements of the first column are stored in contiguous memory locations, followed by all elements of the second column, and so on.

1 2 3

4 5 6

Computer will store like this

1	4
2	5
3	6

0,0	1,0	0,1	1,1	0,2	1,2
1	4	2	5	3	6
300	304	308	312	316	320

=> address[array[I][j]] = baseaddress + w (M (j-lj) + (i-li))

M = number of Row I = row index j = column index

li = starting index of row lj = starting index of column

Q7. Define array. List out all array operations and explain any two with algorithm. (other than search operation).

=> An array is collection of elements which has a same data type that stores data in contiguous memory location.

=> Each element in an array is accessed using an index, with the first element having an index of 0 in most programming languages. Arrays provide a way to

store multiple values in a single variable, making it easier to manage and manipulate collections of data.

=> **Array Operations**

Common operations performed on arrays include:

1. **Traversal**: Accessing each element of the array one by one.
2. **Insertion**: Adding an element at a specified position in the array.
3. **Deletion**: Removing an element from a specified position in the array.
4. **Updation**: Modifying the value of an element at a specified index.
5. **Search**: Finding the index of an element in the array (already discussed).
6. **Sorting**: Arranging the elements of the array in a specific order (e.g., ascending or descending).
7. **Merging**: Combining two or more arrays into one.
8. **Splitting**: Dividing an array into two or more smaller arrays.

=> **Insertion**

Insertion is the process of adding a new element at a specific position in the array. To insert an element, we need to shift all elements from the specified position to the right to make space for the new element.

```
Algorithm Insertion(array, N, position, value) {
        Step 1:  [initialize]  I <= N-1
        Step 2:  Repeat step 3,4 while I >= position
        Step 3: array[i+1] <= array[i]
        Step 4: I <= i-1
                        [End of the loop]
        Step 5: set N <= N+1
        Step 6: set  array[position]  <= value
        Step 7: exit
}
```

=> **Deletion**

Deletion is the process of removing an element from a specific position in the array. After deleting the element, we need to shift all elements from the right of the specified position to the left to fill the gap.

~ Created by Dhruv Prajapati

```
Algorithm Deletion(array, N, position) {
        Step 1: [initialize] I <= position
        Step 2: Repeat steps 3,4 while i<N-1
        step 3: array[I] <= array[I+1]
        step 4: I <= I+1
                        [End Of the Loop]
        step 5: N <= N-1
        step 6: exit
}
```