Data Structure

- A data structure is a group of data elements that provides the easiest way to store and perform different actions on the data of the computer.

- A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.

How Data Structure varies from Data Type

Data Type	Data Structure
• The data type is the form of a variable to which a value can be assigned. It defines that the particular variable will assign the values of the given data type only.	• Data structure is a collection of different kinds of data. That entire data can be represented using an object and can be used throughout the program.
• It can hold value but not data. Therefore, it is dataless.	• It can hold multiple types of data within a single object.
• The implementation of a data type is known as abstract implementation.	• Data structure implementation is known as concrete implementation.
• There is no time complexity	• Time complexity plays an important role.
• In the case of data types, the value of data is not stored because it only represents the type of data that can be stored.	• While in the case of data structures, the data and its value acquire the space in the computer's main memory. Also, a data structure can hold different kinds and types of data within one single object.
• Examples are int, float, double, etc.	• Examples are stack, queue, tree, etc.

Classification of Data Structure

- Data structures can be classified into several categories based on their characteristics and the way they store and organize data. Here are some commonly used classifications of data structures:

- **Primitive Data Structures:** These are the basic data types provided by programming languages, such as integers, floating-point numbers, characters, and booleans. They are not explicitly defined as data structures but serve as the building blocks for more complex data structures.

- **Linear Data Structures:** In linear data structures, data elements are organized sequentially or linearly. Each element has a unique predecessor and successor, except for the first and last elements. Examples of linear data structures include arrays, linked lists, stacks, and queues.

- **Non-linear Data Structures:** Non-linear data structures do not have a sequential arrangement of elements. The relationship between elements can be hierarchical or arbitrary. Examples include trees, graphs, heaps, and hash tables.

Classification of Data Structure

Primitive Data Type

- Primitive data types are the data types available in most programming languages.
- These data types are used to represent single values.
- It is a basic data type available in most programming languages.

Non-Primitive Data Type

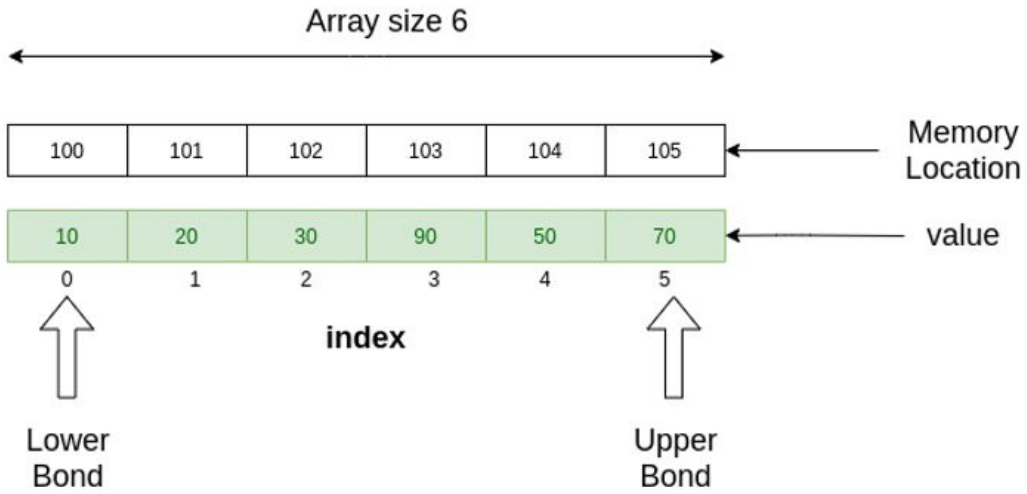
- Data types derived from primary data types are known as Non-Primitive data types.
- Non-Primitive data types are used to store groups of values.

Linear ADTs:

- Linear ADTs organize data elements in a linear or sequential manner, where each element has a unique predecessor and successor, except for the first and last elements.
- The key characteristic of linear ADTs is the linear relationship between the elements. Some examples of linear ADTs are:

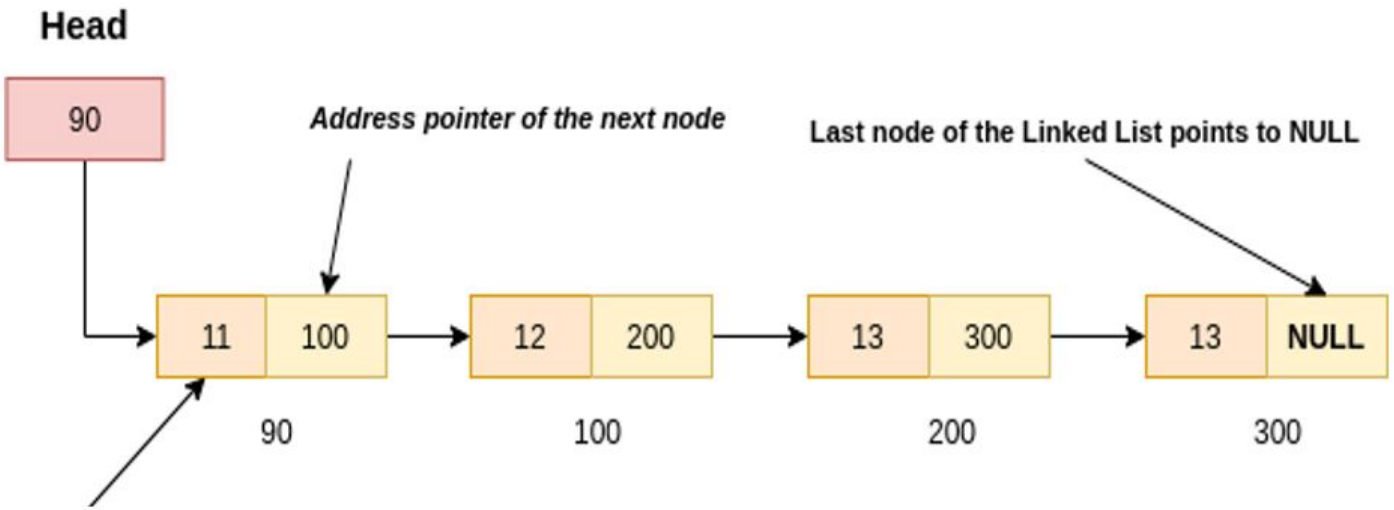
Array

- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Linear ADTs:

- **Lists**: A list is a collection of elements arranged in a specific order. Elements can be accessed sequentially, and new elements can be added or removed from either end or a specific position within the list. Examples of linear lists include arrays and linked lists.



Static Vs Dynamic Data Structures

Specification	Static Data Structure	Dynamic Data Structure
Memory allocation	Memory is allocated at compile-time	Memory is allocated at run-time
Size	Size is fixed and cannot be modified	Size can be modified during runtime
Memory utilization	Memory utilization may be inefficient	Memory utilization is efficient as memory can be reused
Access	Access time is faster as it is fixed	Access time may be slower due to indexing and pointer usage
Examples	Arrays, Stacks, Queues, Trees (with fixed size)	Lists, Trees (with variable size), Hash tables

Operations on different Data Structure

Create

It reserves memory for program elements by declaring them. The creation of data structure can be done during

1. Compile-time
2. Run-time.

- You can use malloc() function.

Selection

- It selects specific data from present data. You can select any specific data by giving condition in loop .

Update

- It updates the data in the data structure. You can also update any specific data by giving some condition in loop like select approach.

Operations on different Data Structure

Sort

- Sorting data in a particular order (ascending or descending).
- Can take the help of many sorting algorithms to sort data in less time.
- Example: bubble sort which takes O(n^2)time to sort data.
- There are many algorithms present like merge sort, insertion sort, selection sort, quick sort, etc.

Merge

- Merging data of two different orders in a specific order may ascend or descend. We use merge sort to merge sort data.

Split Data

- Dividing data into different sub-parts to make the process complete in less time

Algorithms

- Find a method or steps that will help you get your work done, we call this method “Algorithm”
- In simple terms, Algorithm is a step-by-step process followed to do/accomplish any task.
- *An algorithm is a set of well-designed, step-by-step instructions designed to solve a problem or perform a specific task.*
- An algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high-level description as pseudocode or using a flowchart.

Characteristics Of An Algorithm

- Not all procedures can be called an algorithm. An algorithm should have the following characteristics:
- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** will get one or more outputs at the end of an algorithm.
- **Unambiguity**: An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** Algorithms must terminate after a finite number of steps.
- **Effectiveness**: An algorithm should be effective with the available resources.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output

Factors of an algorithm

1. **Correctness:** An algorithm should produce the correct output for each valid input. It should solve the problem it was designed to address accurately.
2. **Efficiency:** Efficiency refers to how well an algorithm utilizes computational resources, such as time and memory. An efficient algorithm typically accomplishes its task in a reasonable amount of time and with minimal resource usage.
3. **Scalability:** Scalability relates to how well an algorithm performs as the input size increases. A scalable algorithm maintains its efficiency even when working with larger datasets or more complex problems.
4. **Complexity:** Algorithmic complexity describes the computational resources required by an algorithm, typically measured in terms of time complexity (how long it takes to run) and space complexity (how much memory it requires).
5. **Optimality:** An optimal algorithm is one that achieves the best possible solution for a given problem. Depending on the problem, optimality may refer to finding the shortest path, the smallest solution, or the highest value, among other criteria.