Data Structures and Algorithms

What are Algorithms ? What are Data Structures?

Week-1

Why study data structures and algorithms?

- People learn by experience.
- Optimize the code to take less or sometime feasible time.

TIME AND SPACE COMPLEXITY

- ★ To get the actual time complexity we will need a lot of things like:
- ➔ The Speed of the Computer.
- ➔ The System Architecture.
- ➔ The Compiler being used.
- ➔ Details of the memory Hierarchy.

- ➔ So, We won't be using them. We will be taking two assumptions:-
  - ◆ Get results that work for large inputs.
  - ◆ Measure runtime without knowing these details.(given above)

Big O,Omega and Theta Notation

- ➔ The Big O notation defines an upper bound of an algorithm
- ➔ The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
- ➔ The Omega notation provides an asymptotic lower bound.
- ➔ Big-oh is the most useful because represents the worst-case behavior. So, it guarantees that the program will terminate within a certain time period, it may stop earlier, but never later.

Not Compulsory to use Big O for worst case,etc

- ➔ Big O Notation is a convenient way to express the worst-case scenario for an algorithm
- ➔ But, there is no relationship of the type “big O is used for worst case, Theta for average case”. All types of notation can be (and sometimes are) used when talking about best, average, or worst case of an algorithm.
- ➔ Worst case can be shown using : big o, theta, or omega
- ➔ 10^5 * N units would have complexity of O(K * N).

Pseudocode:

- ➔ In computer science, pseudocode is an informal high-level description of the operating principle of a computer program.

- ➔ They are partial English and partial code.
- ➔ They do not follow syntax, so don’t try to run them using your own created AI based compiler.

Intuition

A thing that one knows or considers likely from instinctive feeling rather than conscious reasoning.

It is like the first step to design any Data Structure and to design any Algorithm.

We apply Conscious reasoning once we have designed something with Intuition first.

ADT

- ➔ Abstract data types, commonly abbreviated ADTs, are a way of classifying data structures based on how they are used and the behaviors they provide.
- ➔ They do not specify how the data structure must be implemented but simply provide a minimal expected interface and set of behaviors.
- ➔ Data Structure is a concrete implementation of a data type. It’s possible to analyze the time and memory complexity of a Data Structure but not from a data type. The Data Structure can be implemented in several ways and its implementation may vary from language to language.

Linear Data Structures:

- A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element. This connection helps to traverse a linear data structure in a single level and in single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

Arrays-ADT

Array is a container which can hold a fix number of items and these items should be of the same type.They are stored in contiguous memory locations so that accessing any element randomly happens in constant time.Following are the important terms to understand the concept of Array.

Element − Each item stored in an array is called an element.

Index − Each location of an element in an array has a numerical index, which is used to identify the element.

Basic Operations of Array

- ➔ Traverse() – print all the array elements one by one.
- ➔ Insertion() – Adds an element at the given index.
- ➔ Deletion() – Deletes an element at the given index.
- ➔ Search() – Searches an element using the given index or by the value.
- ➔ Update() – Updates an element at the given index.

Dynamic Arrays:

Arrays are usually created with a defined size and you can’t change it after its creation.

Dynamic arrays are implemented in a way they are capable of resizing and holding more elements as needed.

Higher-level languages like Python, Golang, Java and, Javascript implement dynamic arrays.

Linked List-ADT

- ➔ It holds a collection of Nodes, the nodes can be accessed in a sequential way. Linked List doesn’t provide a random access to a Node.
- ➔ When the Nodes are connected with only the next pointer the list is called Singly Linked List and when it’s connected by the next and previous the list is called Doubly Linked List.

Basic Operations of Linked List

- ➔ prepend(value) - Add a node in the beginning
- ➔ append(value) - Add a node in the end
- ➔ pop() - Remove a node from the end
- ➔ popFirst() - Remove a node from the beginning
- ➔ head() - Return the first node
- ➔ tail() - Return the last node
- ➔ remove(Node) - Remove Node from the list

Stack-ADT

Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first.

Basic Operations of Stack

- ➔ push(item) - Adds a new item to the top of the stack. It needs the item and returns nothing.
- ➔ pop() - Removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- ➔ peek() - Returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- ➔ isEmpty() - Tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- ➔ size() - Returns the number of items on the stack. It needs no parameters and returns an integer.

Queue-ADT

- ➔ Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- ➔ In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end.

Basic Operations of Queue

- ➔ enqueue() – Insert an element at the end of the queue.
- ➔ dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- ➔ peek() – Return the element of the queue without removing it, if the queue is not empty.
- ➔ size() – Return the number of elements in the queue.
- ➔ isEmpty() – Return true if the queue is empty, otherwise return false.
- ➔ isFull() – Return true if the queue is full, otherwise return false.