What is Data Structure and Algorithm ?

By – Mohammad Imran

Introduction to Data Structure

By – Mohammad Imran

Introduction

A data structure is a meaningful way of arranging and storing data in a computer so as to use it efficiently.

If you go out shopping at a grocery store, you would expect related items to be in the same section, such as the fruit and vegetable section, the meat section, the dairy section, etc.

These items can be further organized according to their prices and so on.

Data
+
Structure
=
Data Structure

Basic Terminology

Database -

An organized record of data so as to use it efficiently, nevertheless usually stored in hard disk or permanent memory as opposed to data structures being stored usually in RAM or volatile memory.

Algorithm -

Is a step-by-step set of instructions for doing stuff such as making an omelet, playing rugby, checking primes, and reading this article. From washing machines to self-driving cars to every deterministic action ever taken can be expressed as algorithms.

Basic Terminology

Data -

Represents an atomic value or collection of facts that could lead to contextual information e.g. a unit value 40 or a collection such as [(35, 19), (35, 18), (30, 18), (29, 18), (29, 17)] doesn't make sense in isolation but are referred to as data nonetheless. It's only when we associate respective contexts like the price of apples per kg. or 5-day temperature forecast, do we harness information out of the raw data.

Basic Terminology

Asymptotic Complexity -

Determines how fast an algorithm can compute (with respect to input) when applied over a data structure.

Likewise, you wouldn't want to use a slow computer where searching for an old photo takes around a day. That's why we have an efficient data structure called the ‘file system’ that organizes and stores said data into a hierarchy starting from the root directory.

Data Structures

Importance in Programming

Data structures are essential for two main reasons: they make the code more efficient, and they make the code easier to understand. When it comes to efficiency, data structures help the computer to run the code faster by organizing the data in a way that is easy for the computer to process.

In the world of programming, data structures are the backbone of efficient data organization and manipulation. They're like having a well-organized toolbox, where each tool has its specific place and purpose, enabling us to write code that's not just functional but streamlined.

Real World Application

Data structures are a fundamental concept in computer science and have different applications in various fields. Some of the most common applications of data structures .

Here is a list of some common applications of Data structures, along with examples:

Real World Application

- ✓ **Database Management Systems** - Data structures are used to store and organize large amounts of data in a structured manner. For example, a relational database uses tables and indices to store data, and a NoSQL database uses document-oriented or key-value data structures.
- ✓ **Operating Systems** - Data structures are used to manage resources, such as memory allocation and process scheduling. For example, a priority queue data structure can be used to manage the scheduling of processes in an operating system.

Real World Application

- ✓ **Graphical User Interfaces** - Data structures are used to store and display hierarchical data, such as the file system in a computer's file explorer. For example, a tree data structure can be used to represent the file hierarchy in a file explorer.

- ✓ **Computer Networks** - Data structures are used to store information about network topology and to implement routing algorithms. For example, a graph data structure can be used to represent a computer network, with nodes representing devices and edges representing connections between devices.

Real World Application

- ✓ **Compiler Design** - Data structures are used to store and process the syntax and semantics of programming languages. For example, a parse tree data structure can be used to represent the structure of a program and to generate intermediate code.

- ✓ **Artificial Intelligence and Machine Learning** - Data structures are used to store and manipulate large amounts of data in order to train algorithms and make predictions. For example, a decision tree data structure can be used in a machine learning algorithm to make predictions based on input data.

Real World Application

- ✓ **Computer Graphics** - Data structures are used to store and manipulate geometric models for rendering and animation. For example, a 3D mesh data structure can be used to represent a 3D object in computer graphics.
- ✓ **Bioinformatics** - Data structures are used to store and analyze large amounts of biological data, such as DNA sequences and protein structures. For example, a suffix tree data structure can be used to efficiently store and search for patterns in DNA sequences.

Real World Application

- ✓ **Web Search Engines** - Data structures are used to store and retrieve large amounts of web pages and to rank them based on relevance. For example, an inverted index data structure can be used to store keywords and their corresponding web pages, allowing for fast searching and retrieval.

- ✓ **Computer Algorithms and Optimization** - Data structures are used to design and analyze algorithms for solving complex problems. For example, a priority queue data structure can be used in Dijkstra's algorithm for finding the shortest path in a graph.

Real World Application

- ✓ **Geographic Information Systems (GIS)** - Data structures are used to store and analyze geographical data, such as maps, satellite images, and census data. For example, a quadtree data structure can be used to efficiently store and search for geographical features, such as cities or roads.

- ✓ **Video Games** - Data structures are used to store and manipulate game objects and their interactions. For example, a spatial partitioning data structure can be used to efficiently search for and detect collisions between game objects.

Real World Application

- ✓ **Financial Systems** - Data structures are used to store and manage financial data, such as stock prices and transaction history. For example, a balanced binary search tree data structure can be used to efficiently store and retrieve stock prices.
- ✓ **Cryptography** - Data structures are used to store and process encrypted data and to implement cryptographic algorithms. For example, a hash table data structure can be used to store and look up values for encryption and decryption.

Real World Application

- ✓ **Natural Language Processing (NLP)** - Data structures are used to store and process large amounts of text data, such as for text classification, sentiment analysis, and text generation. For example, a trie data structure can be used to efficiently store and search for words in a large text corpus.

Types of Data Structures

There are mainly two types of data structure –

- ✓ Primitive Data Structure
- ✓ Non-Primitive Data Structure

Types of Data Structures

Primitive Data Structure

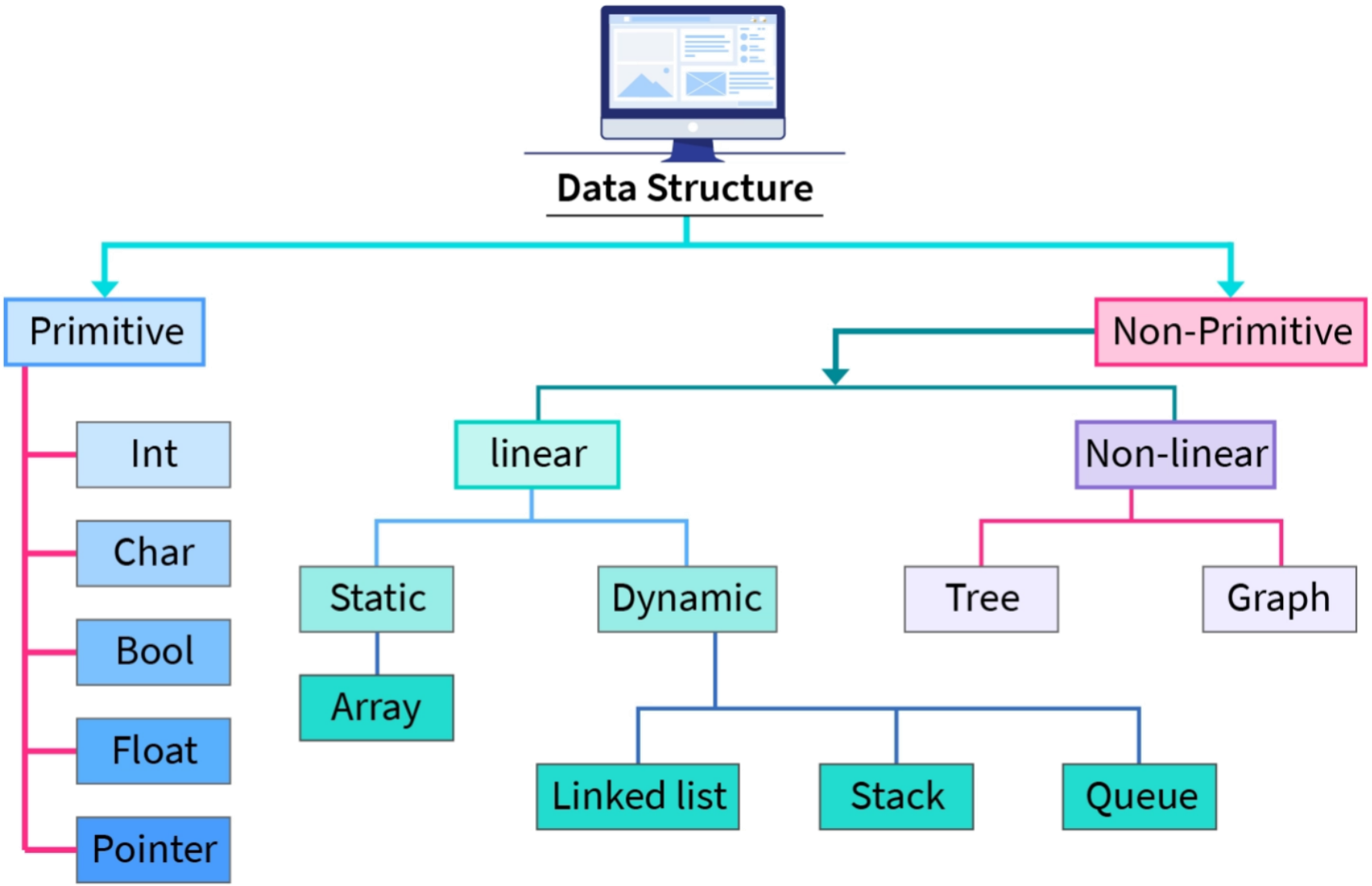
The **primitive data structure**, also known as **built-in data types** can store the data of only one type. You know the integers, floating points, characters, pointers, stuff like that.

Types of Data Structures

Non-Primitive Data Structure

Non-primitive data structures, on the other hand, can store data of more than one type. For example, *array, linked list, stack, queue, tree, graph*, and so on. These are often referred to as **derived data types**.

Data Structures Hierarchy



By – Mohammad Imran

Data Structures

Non - Primitive

As You can see the **Non-Primitive Data Structures** are further classified into two another type of data structure.

- ✓ Linear Data Structure
- ✓ Non-Linear Data Structure

Non – Primitive Data Structures

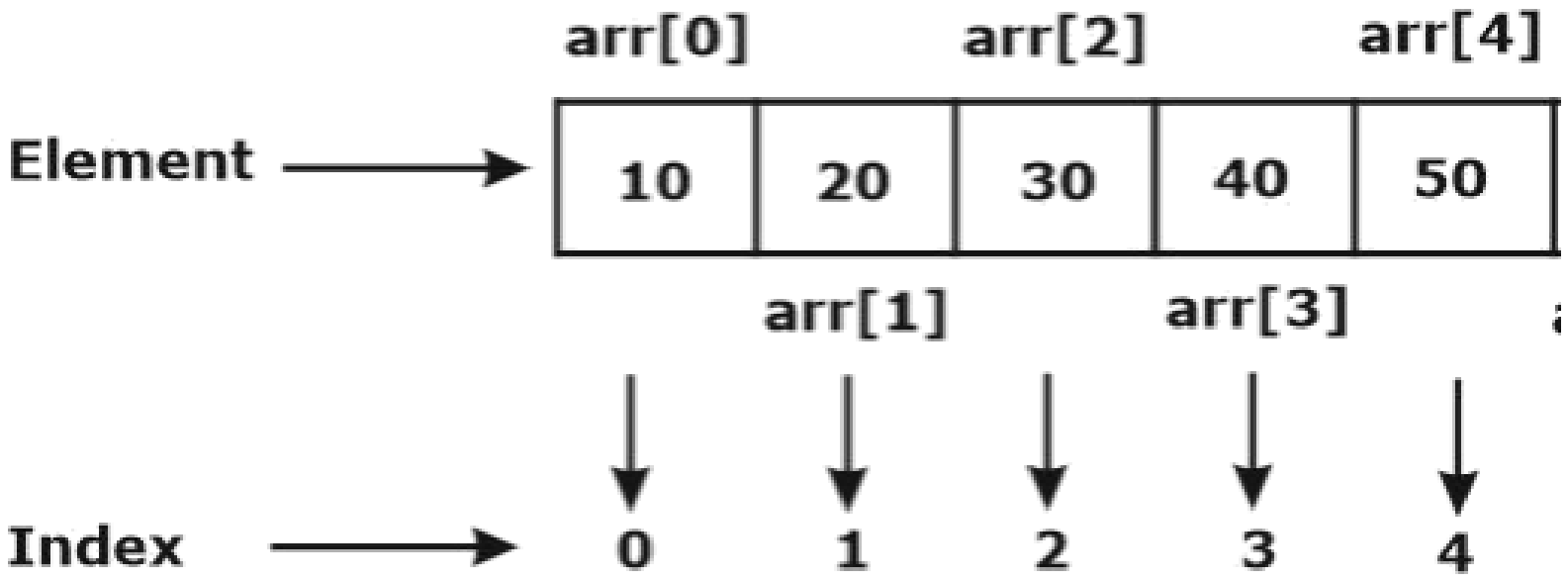
Linear Data Structure

As the name suggests, its data have to be structured in a linear order. That means there is no hierarchy, and elements are held together sequentially either by pointers or contiguous memory locations.

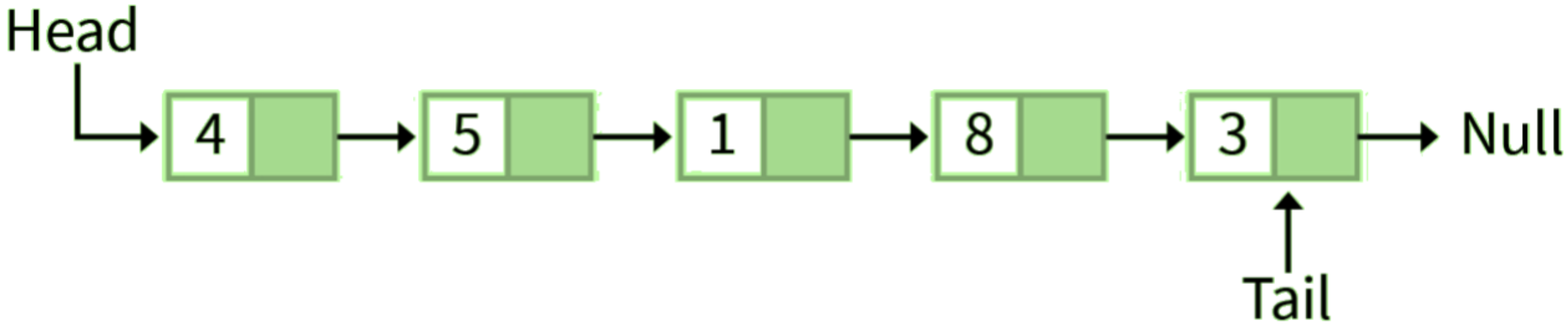
Let's take a scenario, say we want to store the prices of all the groceries that we purchased. So we can allocate a contiguous memory block of size 'n', where 'n' refers to the number of items we purchased.

Array

Here, an array could be a suitable data structure as it's just a collection of similar data types like int, float, char, etc.



Instead of allocating the contiguous memory blocks in advance, how about we **wrap our data in a node that spits out a reference that we can use to point to the next node**.



Linear Data Structures

Some other example of linear data structure is –

- ✓ Stack
- ✓ Queue

Linear Data Structures

Types

Linear data structures are mainly classified into two categories, static and dynamic.

Static Data Structures -

Here the size of the data structure is allocated in the memory during the compile-time thereby rendering the allocated size fixed. This results in the maximum size of such data structures needing to be known in advance, as memory cannot be reallocated at a later point. That’s why they are called static.

Dynamic Data Structure -

Here the size is allocated at the runtime which makes it flexible. As the memory can always be reallocated depending on the requirements, these data structures are called dynamic.

Non - Linear Data Structures

Non-linear data structures are not arranged sequentially in that each element of such data structures can have multiple paths to connect to other elements or form a hierarchy.

Example:

- ✓ Tree
- ✓ Graph

Abstract Data Type (ADT)

In computer science, we often talk about abstraction and programming to an interface.

For instance, you don’t need to know automobile engineering to drive your car to the grocery store. You just have to know how to drive since the manufacturers already abstracted away all the intricacies of the car engine and other internal mechanisms.

Likewise, every data structure has a corresponding interface known as Abstract Data Type or ADT. Simply put, an abstract data type only addresses the interface, and data structures implement that interface.

Abstract Data Type (ADT)

Queue

For instance, a queue is an ADT that must maintain the First In First Out (**FIFO**) ordering of elements. It simply means, the **first** person to get in the queue, would be the **first** person to get **out** of the queue.

This idea of a queue ADT might have been inspired by a real-world standing queue in front of a grocery store counter or similar scenarios.

Abstract Data Type (ADT)

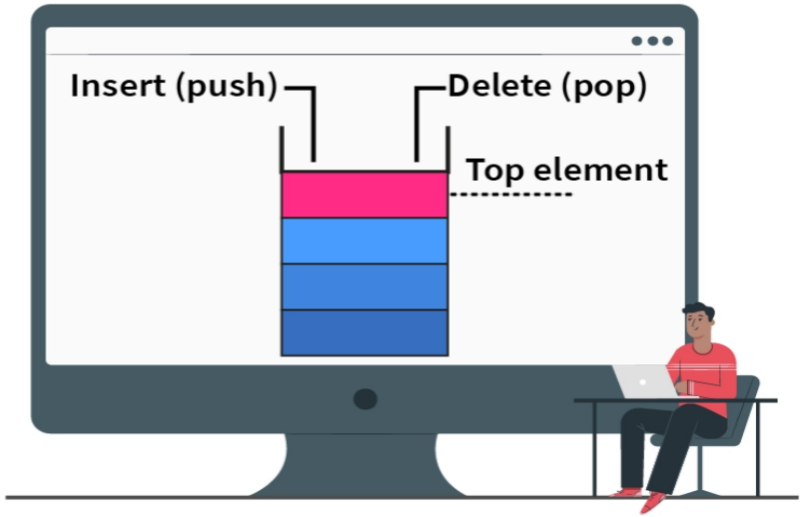
Stack

On the other hand, Stack is another example of ADT that conforms to the Last In First Out (LIFO) ordering of elements.

This idea of this ADT is probably inspired by a real-world stack of books.

Stack of books

Computer stack



Basic Operation

We have covered a lot of ground by discussing several data structures such as array, linked list, stack, queue, tree, graph, hashmap and probably hinted at their respective use-cases but never really got to the specifics.

Well, it turns out almost all data structures support a few key operations... which we are about to explore

Search

Here, an item like Rs. 50 chocolate bar would be your data and the cart itself is a data structure. Let's assume you forgot afterward if you had picked the chocolate bar or not.

How can you ever be sure if you are unable to search?

Traverse

Traversing means iterating through a data structure in some particular order.

In an array, you could access an element in constant time i.e. O(1) provided you knew its position beforehand!

But that doesn't mean you can search for it in constant time.

You still need to start from the first index and search for it. Hence, in order to search you need to traverse. 

So, we conclude that traversal is also as useful as searching.

Insert

Insertion basically means to insert one or more elements into your data structure. Elements can be inserted at the beginning, end or at any specified position.

A cart 🛒 exists for a reason to help facilitate storing items for purchase. In order to store it, you must insert it.

Delete

Once we are done with our shopping, we need to delete items from the cart, show them to the counter, and put it into our bag (another data structure).

Update

This one builds on top of searching and inserting e.g. you suddenly realized you've purchased the wrong chocolate bar so you search the wrong bar in the cart and replace it with the correct bar.

Sort

This is a quite advanced data structure operation and an array is highly efficient for this kind of operation due to its constant access and linear ordering.

Merge

Merging is also used for one of the most well-known sorting algorithms, namely ***merge sort*** (how original).

Merging is not a fundamental operation.

Advantages

None of my previous rants would make sense if there weren't inherent benefits for using a data structure.

But I must tell you that these advantages are generic and conformed by every single data structure, so let's check them out:

- ✓ **Efficiency** - All these various data structures exist for a reason and that is to facilitate efficient storage and retrieval of data for various use-cases.

A tree is more likely to be efficient to model any file system but not efficient for representing relations in social media applications.

Advantages

- ✓ **Abstractions** - Every data structure provides clean interfaces in the form of operations exclusive to their respective abstract data types, which makes their internal implementation hidden from developers.

You don't need to know how STL's *unordered_map* works under the hood to write an application using it in C++, you just need to know what operations it supports and how you can use that to your advantage?

Advantages

- ✓ **Composition** - Fundamental data structures can be combined to build more niche and complex data structures.

In a database management system, indexing is usually implemented using a **B+ tree** which is based on top of the **B tree** - a special kind of n-array self-balancing tree data structure.

In fact, you can think of the database itself as this huge composite data structure capable of storing data even when a program has run its course.

Algorithm

By – Mohammad Imran

Algorithm

Introduction

An algorithm is a set of rules or instructions used to solve a specific computational problem or achieve a desired outcome. In the context of data structures, an algorithm refers to a procedure for manipulating or processing data stored in a particular data structure such as an array, list, tree, graph, etc.

Technically speaking, an algorithm is defined as a set of rules or a step-by-step procedure that are to be executed in a specific order to get the desired output.

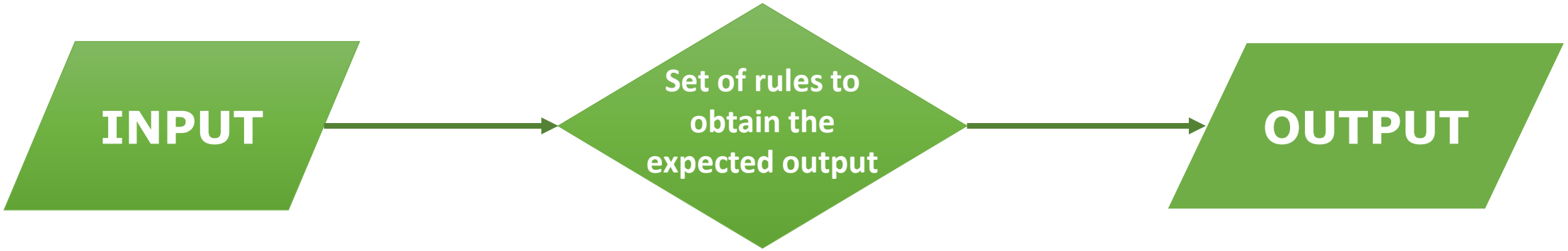
Algorithm

Introduction

Algorithms can be designed using various computational paradigms, such as divide and conquer, greedy, dynamic programming, and others. They can be expressed using pseudocode or a high-level programming language. The efficiency and effectiveness of an algorithm are determined by its time complexity and space complexity, which describe the amount of time and memory required to execute the algorithm, respectively.

Algorithm

Definition

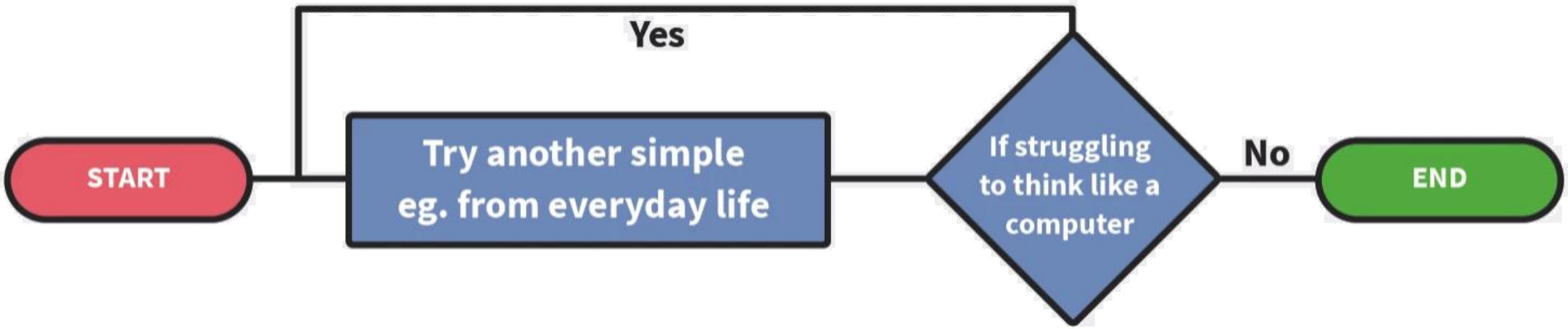


The formal definition of an algorithm is a finite set of steps carried out in a specific time for specific problem-solving operations, especially by a Computer.

Algorithms are independent of programming languages and are usually represented by using flowcharts or pseudocode.

Algorithm Characteristics

A fact to be noted is that not all sets of instructions or procedures are an algorithm. A set of instructions should have all the relevant characteristics to qualify as an algorithm.



Algorithm

Characteristics

The characteristics of algorithm are as follows-

- ✓ **Unambiguous**– An algorithm should be clear and simple in nature and lead to a meaningful outcome i.e. they should be unambiguous.
- ✓ **Input**– It should have some input values.
- ✓ **Output**– Every algorithm should have well-defined outputs
- ✓ **Finiteness**– The steps of an algorithm should be countable and it should terminate after the specified number of steps.
- ✓ **Effectiveness**– Each step of the algorithm should be effective and efficient enough to produce results. The effectiveness of an algorithm can be evaluated with the help of two important parameters-

Algorithm

Characteristics

- ✓ **Time Complexity**-It is nothing but the amount of time taken by the computer to run the algorithm. We can also call it the computational complexity of an algorithm. It can either be best-case, average-case or worst-case. We always aim for the best-case for effectiveness.
- ✓ **Space Complexity**-It refers to the amount of computational memory needed to solve an instance of the problem statement. The lower the space complexity of an algorithm, the faster the algorithm will work.
- ✓ **Language Independent**– Algorithms should be language independent. In other words, the algorithm should work for all programming languages and give the same output.

Algorithm

Data Flow

- ✓ **Problem –** The problem can be any real world or a programmable problem. The problem statement usually gives the programmer an idea of the issue at hand, the available resources and the motivation to come with a plan to solve it.

Algorithm

Data Flow

- ✓ **Algorithm –** After analyzing the problem, the programmer designs the step by step procedure to solve the problem efficiently. This procedure is the algorithm.
- ✓ **Input –** The algorithm is designed and the relevant inputs are supplied.
- ✓ **Processing Unit –** The processing unit receives these inputs and processes them as per the designed algorithm
- ✓ **Output –** Finally, after the processing is complete, we receive the favorable output of our problem statement.

Need

Primarily, we need algorithms for the following two reasons-

- ✓ **Scalability–** When we have big real-world problems, we cannot tackle them on the macro level. We need to break them down into smaller steps so that the problem can be analyzed easily. Thus, algorithms facilitate scalability.
- ✓ **Performance–** It is never easy to break down big problems into smaller modules. But algorithms help us achieve this. They help us make the problem feasible and provide efficient performance driven solutions

Algorithm

How To Write

Different person would tell you that there is no one “right way” to bake a cake. There are countless recipes available online - devised and tested by bakers around the world. Likewise, there are no predefined standards on how to write an algorithm.

The way we write algorithms is heavily influenced by the problem statement and the resources available. The common construct that is widely followed in case of algorithms is the use of pseudocode.

Algorithm

Factors

While designing an algorithm, we must consider the following factors –

- ✓ **Modularity-** If a big problem can be easily broken down into smaller ones, it facilitates modularity.
- ✓ **Correctness-** The analysis of the problem statement and consequently the algorithm should be correct. The algorithm should also work correctly for all possible test cases of the problem at hand.
- ✓ **Maintainability-** The algorithm should be designed in such a way that it should be easy to maintain and if we wish to refine it at some point.

Algorithm

Factors

- ✓ **Functionality-** The steps of an algorithm should successfully solve a real world problem.
- ✓ **User-friendly-** It should be easily understood by programmers.
- ✓ **Simplicity-** It should be the simplest possible solution to a problem. By simplicity, we refer to the fact that the algorithm should have the best-case time complexity.
- ✓ **Extensible-** It should be extensible i.e. the algorithm should facilitate reusability.

Example – 1

Example 1: Design an algorithm to accept three numbers and print their sum.

- ✓ Step 1-**START**
- ✓ Step 2- Declare variables **x,y,z** and **sum**
- ✓ Step 3- Read the value of **x,y** and **z**
- ✓ Step 4- Add **x,y** and **z**
- ✓ Step 5-Store the output of Step 4 in **sum**
- ✓ Step 6-Print **sum**
- ✓ Step 7- **STOP**

Importance

The importance of algorithms can be classified as-

- ✓ **Theoretical Importance-** The best way to deal with any real-world problem is to break them down into smaller modules. The theoretical knowledge from pre-existing algorithms often help us to do so.

- ✓ **Practical Importance-** Just designing an algorithm theoretically or using some aspects of pre-existing ones is not enough. The real-world problems can only be considered to be solved if we manage to get practical results from it.

Algorithm

Types

An algorithm is a sequential series of steps and processes for solving a problem. While there can be **different types** of algorithms for solving different problems, we consider the following algorithms to be important in programming.

As a starting point, here are the types of Algorithms

Algorithm

Types

Here are some examples of algorithms commonly used in data structures:

- ✓ **Search Algorithms:** Algorithms used to search for specific data in a data structure, such as linear search and binary search.
- ✓ **Sorting Algorithms:** Algorithms used to sort data in a particular order, such as bubble sort, insertion sort, quick sort, and merge sort.
- ✓ **Tree Algorithms:** Algorithms used to manipulate tree data structures, such as breadth-first search and depth-first search.

Algorithm

Types

- ✓ **Graph Algorithms:** Algorithms used to manipulate graph data structures, such as Dijkstra's shortest path algorithm and Kruskal's minimum spanning tree algorithm.
- ✓ **Hash Table Algorithms:** Algorithms used to implement hash tables, a data structure that maps keys to values using a hash function.

Linear Search

The linear search algorithm is also known as a sequential search algorithm and it is the **simplest** of all search algorithms. The linear search involves traversing the list completely and matching each element with the item whose location you are looking for. An item's location is returned if a match is found; otherwise, **NULL** is returned.

Binary Search

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

Bubble Sort

In **Bubble Sort**, an element is swapped repeatedly until it is in the sorted or intended order. The movement of array elements is similar to the movement of air bubbles in water, which is why it is called bubble sort. Similar to bubbles in **water rising** to the surface, array elements in bubble sort move to the end each time they are iterated.

Insertion Sort

Insertion sort involves placing an unsorted item at its proper place in each iteration of a **sorting algorithm**. It is similar to how we sort the cards in our hands in a card game.

First, we select an unsorted card, assuming that the first card is already sorted. The unsorted card should be put on the right if it is greater than the card in hand, otherwise, it should be placed on the left. A similar process is followed in sorting other unsorted cards.

Quick Sort

The **Quicksort algorithm** involves divide-and-conquer. In this method, one element is selected as the pivot, and the others are divided into two sub-arrays based on whether they are less or greater than the pivot. This method is therefore sometimes referred to as partition-exchange sort. This can be done in place, with only a little amount of additional RAM required for sorting..

Selection Sort

Despite its simplicity and **straight forward** nature, selection sort is a powerful sorting algorithm. By comparing the items in place, selection-based sorting divides the list into two sections, left and right of the sorted items. There is no data in the sorted section, while all the data is in the unsorted section. A small list can be sorted using selection sort.

Merge Sort

Basically, **merge sort** splits the input into two halves, repeats the process on both halves, and merges the sorted halves together. From top to bottom, the algorithm divides the list into progressively smaller pieces until only the individual elements remain. It then moves back up to ensure that the merged lists are sorted.

Algorithm Types

Recursive Algorithm

Recursion is the basis for this kind of algorithm. Using recursion, a problem is **broken into sub-problems** and calls itself repeatedly until an underlying condition is able to solve it.

Divide And Conquer Algorithm

The divide and conquer algorithm divides the problem into two sections, the first section dividing it into sub problems of the same type. Section **two involves** solving smaller problems independently, combining the results, and calculating the final answer.

Algorithm Types

Dynamic Programming Algorithms

An algorithm of this type is also referred to as a memoization algorithm because it stores the previously calculated result in order to avoid recalculating it. In dynamic programming, we divide complex problems into smaller overlapping subproblems and store the results for later use in Dynamic Programming.

Algorithm Types

Greedy Algorithm

A greedy algorithm builds solution part by part. In order to decide which part to proceed to next, we look at the benefit it provides immediately. Previous decisions are never taken into account.

Hashing Algorithm

A hashing algorithm operates the same as a searching algorithm, but it includes an index with an ID. We assign a key to specific data when we hash it.

Algorithm Types

Backtracking Algorithm

By using **backtracking** algorithm, problems are solved incrementally, i.e. they are solved recursively by adding pieces one at a time, and removing the solutions that do not meet the constraints of the problem at any point in time.

Coding Questions

By – Mohammad Imran

Question - 1

Create a class in Java to create an array with n element. Enter a number to search the element whether exist or not if exist display the index of the element.

Scenario Question:

You are developing a simple program for a company that tracks employee IDs. The program will allow the user to create an array of employee IDs, input a specific ID to search for, and if that ID exists in the array, the program will display the index where the ID is located. If the ID doesn't exist, it will inform the user that the ID is not present.

Click here to see code

By – Mohammad Imran

Question - 2

Write a Java program to find the maximum of given three numbers using ternary operator in a single line.

Click here to see code

By – Mohammad Imran