An Introduction to Data Structures

Since the invention of computers, people have been using the term "**Data**" to refer to Computer Information, either transmitted or stored. However, there is data that exists in order types as well. Data can be numbers or texts written on a piece of paper, in the form of bits and bytes stored inside the memory of electronic devices, or facts stored within a person's mind. As the world started modernizing, this data became a significant aspect of everyone's day-to-day life, and various implementations allowed them to store it differently.

Data is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values. The data items are then classified into sub-items, which is the group of items that are not known as the simple primary form of the item.

Let us consider an example where an employee name can be broken down into three sub-items: First, Middle, and Last. However, an ID assigned to an employee will generally be considered a single item.

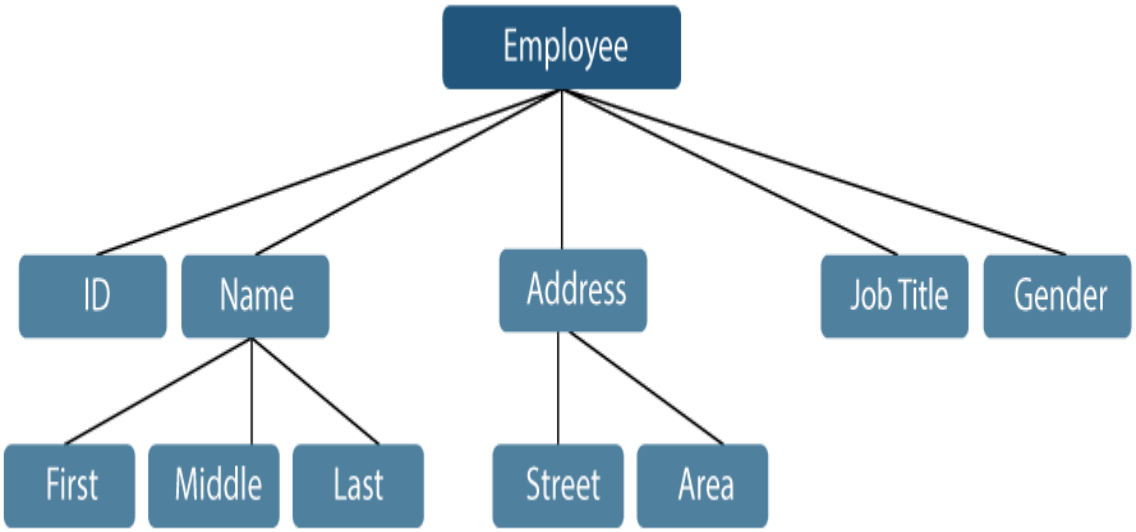

Figure 1: Representation of Data Items

In the example mentioned above, the items such as ID, Age, Gender, First, Middle, Last, Street, Locality, etc., are elementary data items. In contrast, the Name and the Address are group data items.

What is Data Structure?

Data Structure is a branch of Computer Science. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program. Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required. The data can be managed in various ways, like the logical or mathematical model for a specific organization of data is known as a data structure.

The scope of a particular data model depends on two factors:

1. First, it must be loaded enough into the structure to reflect the definite correlation of the data with a real-world object.
2. Second, the formation should be so straightforward that one can adapt to process the data efficiently whenever necessary.

Some examples of Data Structures are Arrays, Linked Lists, Stack, Queue, Trees, etc. Data Structures are widely used in almost every aspect of Computer Science, i.e., Compiler Design, Operating Systems, Graphics, Artificial Intelligence, and many more.

Data Structures are the main part of many Computer Science Algorithms as they allow the programmers to manage the data in an effective way. It plays a crucial role in improving the performance of a program or software, as the main objective of the software is to store and retrieve the user's data as fast as possible.

Basic Terminologies related to Data Structures

Data Structures are the building blocks of any software or program. Selecting the suitable data structure for a program is an extremely challenging task for a programmer.

The following are some fundamental terminologies used whenever the data structures are involved:

1. **Data:** We can define data as an elementary value or a collection of values. For example, the Employee's name and ID are the data related to the Employee.
2. **Data Items:** A Single unit of value is known as Data Item.
3. **Group Items:** Data Items that have subordinate data items are known as Group Items. For example, an employee's name can have a first, middle, and last name.
4. **Elementary Items:** Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.

5. **Entity and Attribute:** A class of certain objects is represented by an Entity. It consists of different Attributes. Each Attribute symbolizes the specific property of that Entity. For example,

Attributes	ID	Name	Gender	Job Title
Values	1234	Stacey M. Hill	Female	Software Developer

Entities with similar attributes form an **Entity Set**. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the specific attribute.

The term "information" is sometimes utilized for data with given attributes of meaningful or processed data.

1. **Field:** A single elementary unit of information symbolizing the Attribute of an Entity is known as Field.
2. **Record:** A collection of different data items are known as a Record. For example, if we talk about the employee entity, then its name, id, address, and job title can be grouped to form the record for the employee.
3. **File:** A collection of different Records of one entity type is known as a File. For example, if there are 100 employees, there will be 25 records in the related file containing data about each employee.

Understanding the Need for Data Structures

As applications are becoming more complex and the amount of data is increasing every day, which may lead to problems with data searching, processing speed, multiple requests handling, and many more. Data Structures support different methods to organize, manage, and store data efficiently. With the help of Data Structures, we can easily traverse the data items. Data Structures provide Efficiency, Reusability, and Abstraction.

Why should we learn Data Structures?

1. Data Structures and Algorithms are two of the key aspects of Computer Science.

2. Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
3. Learning Data Structures and Algorithms will help us become better Programmers.
4. We will be able to write code that is more effective and reliable.
5. We will also be able to solve problems more quickly and efficiently.

Understanding the Objectives of Data Structures

Data Structures satisfy two complementary objectives:

1. **Correctness:** Data Structures are designed to operate correctly for all kinds of inputs based on the domain of interest. In order words, correctness forms the primary objective of Data Structure, which always depends upon the problems that the Data Structure is meant to solve.
2. **Efficiency:** Data Structures also requires to be efficient. It should process the data quickly without utilizing many computer resources like memory space. In a real-time state, the efficiency of a data structure is a key factor in determining the success and failure of the process.

Understanding some Key Features of Data Structures

Some of the Significant Features of Data Structures are:

1. **Robustness:** Generally, all computer programmers aim to produce software that yields correct output for every possible input, along with efficient execution on all hardware platforms. This type of robust software must manage both valid and invalid inputs.
2. **Adaptability:** Building software applications like Web Browsers, Word Processors, and Internet Search Engine include huge software systems that require correct and efficient working or execution for many years. Moreover, software evolves due to emerging technologies or ever-changing market conditions.
3. **Reusability:** The features like Reusability and Adaptability go hand in hand. It is known that the programmer needs many resources to build any software, making

it a costly enterprise. However, if the software is developed in a reusable and adaptable way, then it can be applied in most future applications. Thus, by executing quality data structures, it is possible to build reusable software, which appears to be cost-effective and timesaving.

Classification of Data Structures

A Data Structure delivers a structured set of variables related to each other in various ways. It forms the basis of a programming tool that signifies the relationship between the data elements and allows programmers to process the data efficiently.

We can classify Data Structures into two categories:

1. Primitive Data Structure
2. Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.

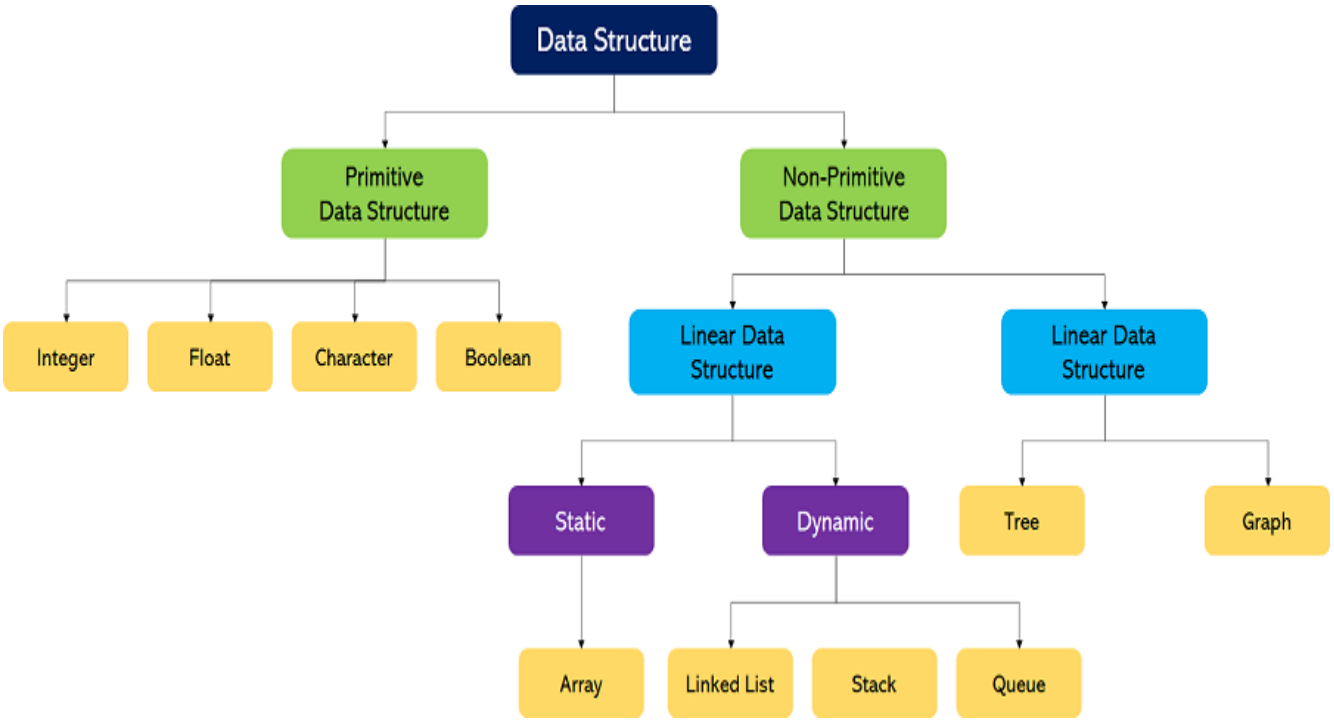

Figure 2: Classifications of Data Structures

Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -

    a. Linear Data Structures
    b. Non-Linear Data Structures

Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the

compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.

The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.

2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.

   Linked Lists, Stacks, and **Queues** are common examples of dynamic data structures

Types of Linear Data Structures

The following is the list of Linear Data Structures that we generally use:

1. Arrays

An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable. This statement doesn't imply that we will have to unite all the values of the same data type in any program into one array of that data type. But there will often be times when some specific variables of the same data types are all related to one another in a way appropriate for an array.

An Array is a list of elements where each element has a unique place in the list. The data elements of the array share the same variable name; however, each carries a different index number called a subscript. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.

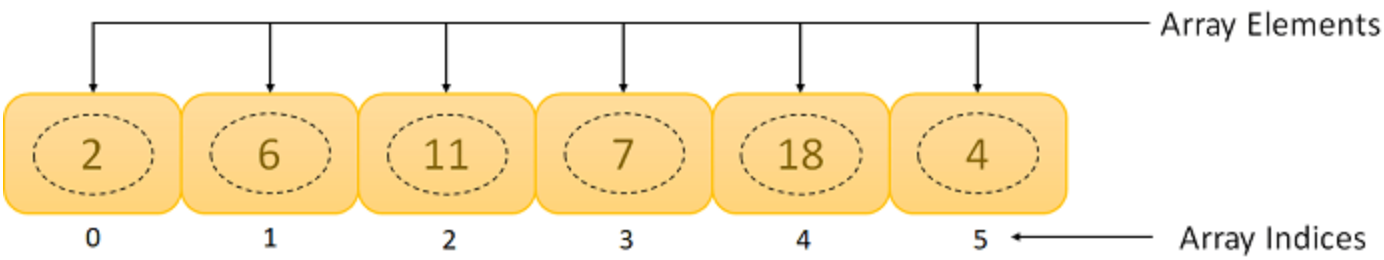

Figure 3. An Array

Arrays can be classified into different types:

a. **One-Dimensional Array:** An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.

b. **Two-Dimensional Array:** An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.

c. **Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

Some Applications of Array:

a. We can store a list of data elements belonging to the same data type.

b. Array acts as an auxiliary storage for other data structures.

c. The array also helps store data elements of a binary tree of the fixed count.

d. Array also acts as a storage of matrices.

2. Linked Lists

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

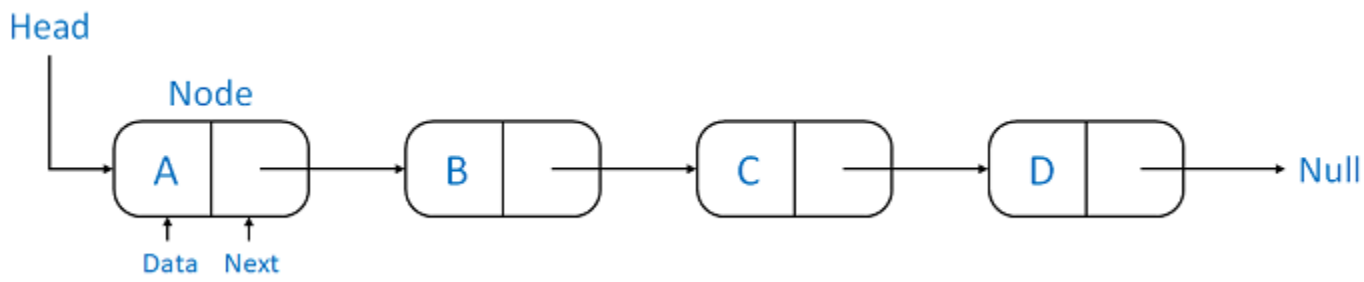

Figure 4. A Linked List

Linked Lists can be classified into different types:

a. **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

b. **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).

c. **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

Some Applications of Linked Lists:

a. The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.

b. We can also implement Operating System's function for dynamic memory management.

c. Linked Lists also allow polynomial implementation for mathematical operations.

d. We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.

e. Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.

f. Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

3. Stacks

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

Figure 5. A Real-life Example of Stack

The above figure represents the real-life example of a Stack where the operations are performed from one end only, like the insertion and removal of new books from the top of the Stack. It implies that the insertion and deletion in the Stack can be done only from the top of the Stack. We can access only the Stack's tops at any given time.

The primary operations in the Stack are as follows:

a. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.

b. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.

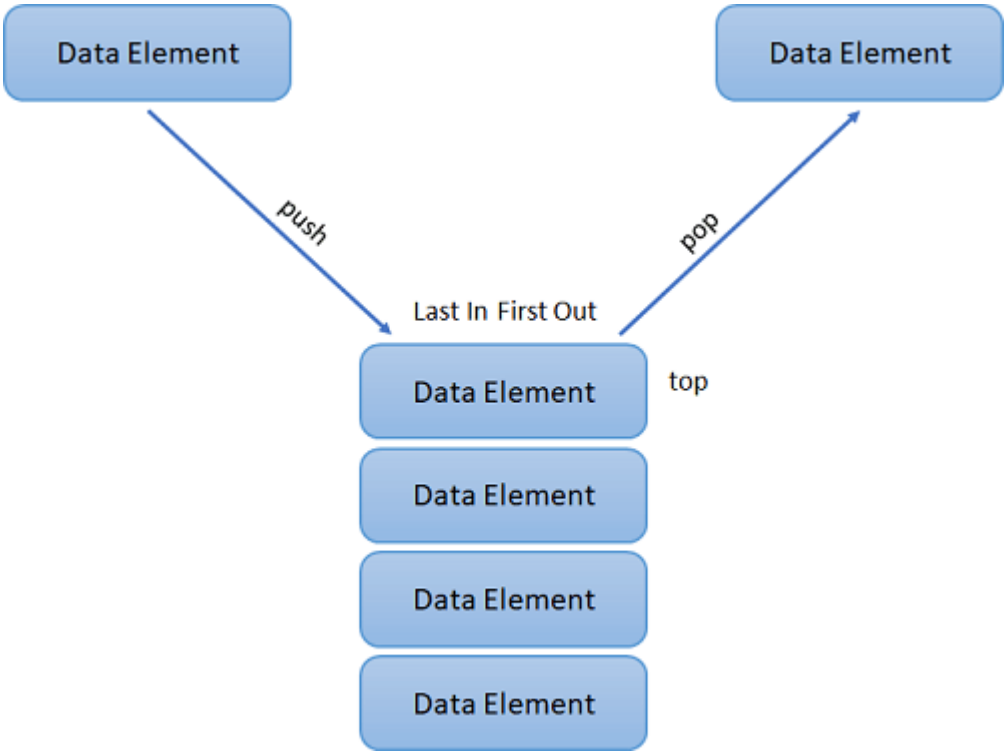

Figure 6. A Stack

Some Applications of Stacks:

a. The Stack is used as a Temporary Storage Structure for recursive operations.

b. Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.

c. We can manage function calls using Stacks.

d. Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.

e. Stacks are also helpful in converting infix expressions to postfix expressions.

f. Stacks allow us to check the expression's syntax in the programming environment.

g. We can match parenthesis using Stacks.

h. Stacks can be used to reverse a String.

i. Stacks are helpful in solving problems based on backtracking.

j. We can use Stacks in depth-first search in graph and tree traversal.

k. Stacks are also used in Operating System functions.

l. Stacks are also used in UNDO and REDO functions in an edit.

4. Queues

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

Figure 7. A Real-life Example of Queue

The above image is a real-life illustration of a movie ticket counter that can help us understand the Queue where the customer who comes first is always served first. The customer arriving last will undoubtedly be served last. Both ends of the Queue are open and can execute different operations. Another example is a food court line where the customer is inserted from the rear end while the customer is removed at the front end after providing the service they asked for.

The following are the primary operations of the Queue:

a. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.

b. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.

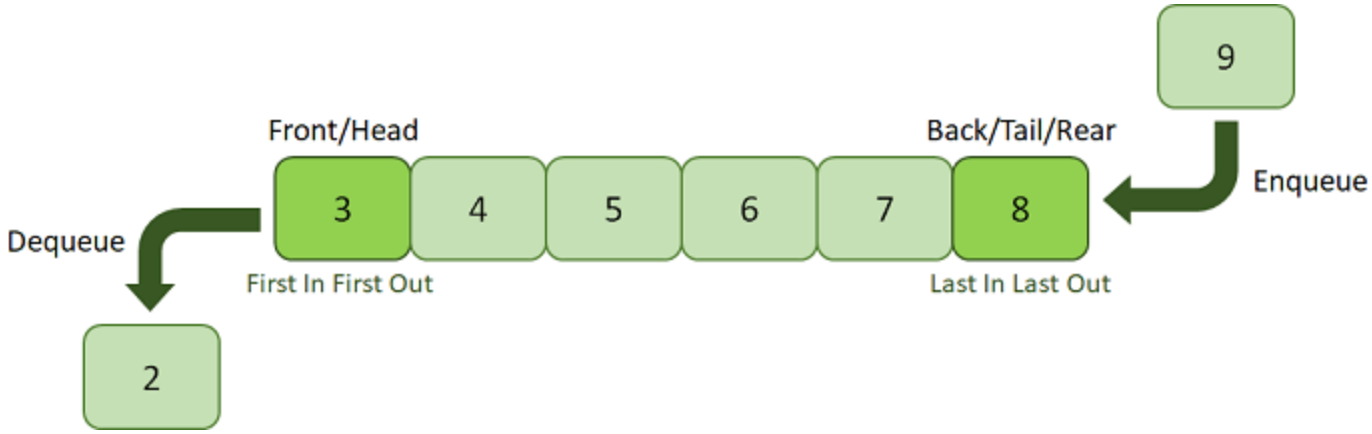

Figure 8. A Queue

Some Applications of Queues:

a. Queues are generally used in the breadth search operation in Graphs.

b. Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.

c. Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.

d. Priority Queues are utilized in file-downloading operations in a browser.

e. Queues are also used to transfer data between peripheral devices and the CPU.

f. Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

Types of Non-Linear Data Structures

The following is the list of Non-Linear Data Structures that we generally use:

1. Trees

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.

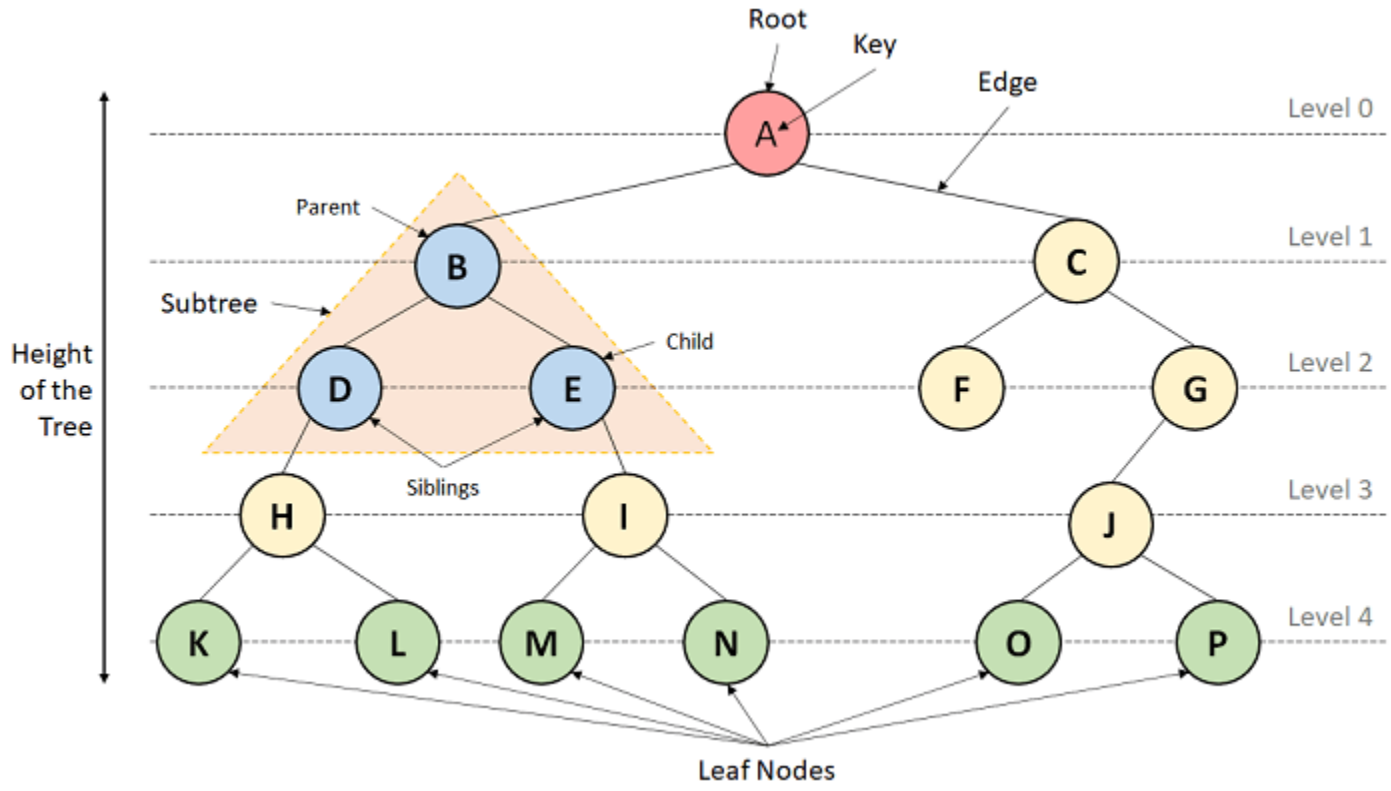

Figure 9. A Tree

Trees can be classified into different types:

a. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.

b. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.

c. **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.

d. **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

Some Applications of Trees:

a. Trees implement hierarchical structures in computer systems like directories and file systems.

b. Trees are also used to implement the navigation structure of a website.

c. We can generate code like Huffman's code using Trees.

d. Trees are also helpful in decision-making in Gaming applications.

e. Trees are responsible for implementing priority queues for priority-based OS scheduling functions.

f. Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.

g. We can use Trees to store data keys for indexing for Database Management System (DBMS).

h. Spanning Trees allows us to route decisions in Computer and Communications Networks.

i. Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.

2. Graphs

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or

vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:

G = (V,E)

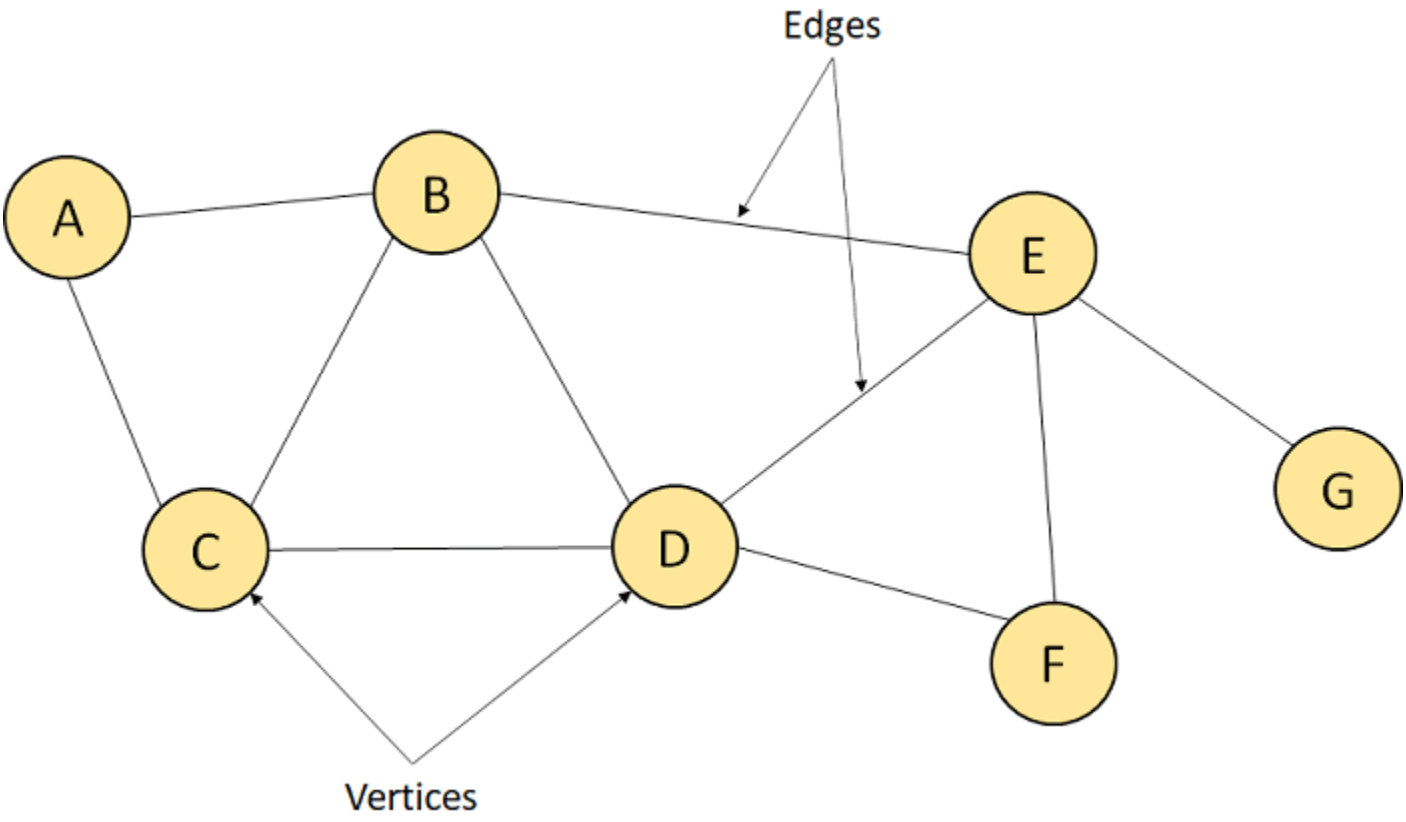

Figure 10. A Graph

The above figure represents a Graph having seven vertices A, B, C, D, E, F, G, and ten edges [A, B], [A, C], [B, C], [B, D], [B, E], [C, D], [D, E], [D, F], [E, F], and [E, G].

Depending upon the position of the vertices and edges, the Graphs can be classified into different types:

a. **Null Graph:** A Graph with an empty set of edges is termed a Null Graph.

b. **Trivial Graph:** A Graph having only one vertex is termed a Trivial Graph.

c. **Simple Graph:** A Graph with neither self-loops nor multiple edges is known as a Simple Graph.
d. **Multi Graph:** A Graph is said to be Multi if it consists of multiple edges but no self-loops.
e. **Pseudo Graph:** A Graph with self-loops and multiple edges is termed a Pseudo Graph.
f. **Non-Directed Graph:** A Graph consisting of non-directed edges is known as a Non-Directed Graph.
g. **Directed Graph:** A Graph consisting of the directed edges between the vertices is known as a Directed Graph.
h. **Connected Graph:** A Graph with at least a single path between every pair of vertices is termed a Connected Graph.
i. **Disconnected Graph:** A Graph where there does not exist any path between at least one pair of vertices is termed a Disconnected Graph.
j. **Regular Graph:** A Graph where all vertices have the same degree is termed a Regular Graph.
k. **Complete Graph:** A Graph in which all vertices have an edge between every pair of vertices is known as a Complete Graph.
l. **Cycle Graph:** A Graph is said to be a Cycle if it has at least three vertices and edges that form a cycle.
m. **Cyclic Graph:** A Graph is said to be Cyclic if and only if at least one cycle exists.
n. **Acyclic Graph:** A Graph having zero cycles is termed an Acyclic Graph.
o. **Finite Graph:** A Graph with a finite number of vertices and edges is known as a Finite Graph.
p. **Infinite Graph:** A Graph with an infinite number of vertices and edges is known as an Infinite Graph.
q. **Bipartite Graph:** A Graph where the vertices can be divided into independent sets A and B, and all the vertices of set A should only be connected to the vertices present in set B with some edges is termed a Bipartite Graph.
r. **Planar Graph:** A Graph is said to be a Planar if we can draw it in a single plane with two edges intersecting each other.

s. **Euler Graph:** A Graph is said to be Euler if and only if all the vertices are even degrees.

t. **Hamiltonian Graph:** A Connected Graph consisting of a Hamiltonian circuit is known as a Hamiltonian Graph.

Some Applications of Graphs:

a. Graphs help us represent routes and networks in transportation, travel, and communication applications.

b. Graphs are used to display routes in GPS.

c. Graphs also help us represent the interconnections in social networks and other network-based applications.

d. Graphs are utilized in mapping applications.

e. Graphs are responsible for the representation of user preference in e-commerce applications.

f. Graphs are also used in Utility networks in order to identify the problems posed to local or municipal corporations.

g. Graphs also help to manage the utilization and availability of resources in an organization.

h. Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.

i. Graphs are also used in robotic motions and neural networks.

Basic Operations of Data Structures

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure:

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For

example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.

3. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.
5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:

    a. Compile-time
    b. Run-time
       For example, the **malloc()** function is used in C Language to create data structure.

2. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
3. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
4. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

Understanding the Abstract Data Type

As per the **National Institute of Standards and Technology (NIST)**, a data structure is an arrangement of information, generally in the memory, for better algorithm efficiency. Data Structures include linked lists, stacks, queues, trees, and dictionaries. They could also be a theoretical entity, like the name and address of a person.

From the definition mentioned above, we can conclude that the operations in data structure include:

a. A high level of abstractions like addition or deletion of an item from a list.
b. Searching and sorting an item in a list.
c. Accessing the highest priority item in a list.

Whenever the data structure does such operations, it is known as an **Abstract Data Type (ADT)**.

We can define it as a set of data elements along with the operations on the data. The term "abstract" refers to the fact that the data and the fundamental operations defined on it are being studied independently of their implementation. It includes what we can do with the data, not how we can do it.

An ADI implementation contains a storage structure in order to store the data elements and algorithms for fundamental operation. All the data structures, like an array, linked list, queue, stack, etc., are examples of ADT.

Understanding the Advantages of using ADTs

In the real world, programs evolve as a consequence of new constraints or requirements, so modifying a program generally requires a change in one or multiple data structures. For example, suppose we want to insert a new field into an employee's record to keep track of more details about each employee. In that case, we can improve the efficiency of the program by replacing an Array with a Linked structure. In such a situation, rewriting every procedure that utilizes the modified structure is unsuitable. Hence, a better alternative is to separate a data structure from its implementation information. This is the principle behind the usage of Abstract Data Types (ADT).

Some Applications of Data Structures

The following are some applications of Data Structures:

1. Data Structures help in the organization of data in a computer's memory.
2. Data Structures also help in representing the information in databases.

3. Data Structures allows the implementation of algorithms to search through data (For example, search engine).
4. We can use the Data Structures to implement the algorithms to manipulate data (For example, word processors).
5. We can also implement the algorithms to analyse data using Data Structures (For example, data miners).
6. Data Structures support algorithms to generate the data (For example, a random number generator).
7. Data Structures also support algorithms to compress and decompress the data (For example, a zip utility).
8. We can also use Data Structures to implement algorithms to encrypt and decrypt the data (For example, a security system).
9. With the help of Data Structures, we can build software that can manage files and directories (For example, a file manager).
10. We can also develop software that can render graphics using Data Structures. (For example, a web browser or 3D rendering software).

Apart from those, as mentioned earlier, there are many other applications of Data Structures that can help us build any desired software.