Data Structure

Chapter 1: Introduction

A **data structure** is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

1.1 Data type

In the C programming language, data types are declarations for memory locations or variables that determine the characteristics of the data that may be stored and the methods (operations) of processing that are permitted involving them.

1.1.1Primitive data types

In computer science, primitive data type is either of the following

- a basic type is a data type provided by a programming language as a basic building block. Most languages allow more complicated composite types to be recursively constructed starting from basic types.
- a built- in type is a data type for which the programming language provides built- in support.

basic primitive types may include:

1. Character (character, char);
2. Integer (integer, int, short, long, byte) with a variety of precisions;
3. Floating- point number (float, double, real, double precision);
4. Fixed- point number (fixed) with a variety of precisions and a programmer-selected scale.
5. Boolean, logical values true and false.

1.1.2Non- Primitive data types

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built- in data types and associated operations on them.

For example – List,Array,Stack,Queue,Structure,class

1.2Abstract data type

An abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.

1.3Need of data structure

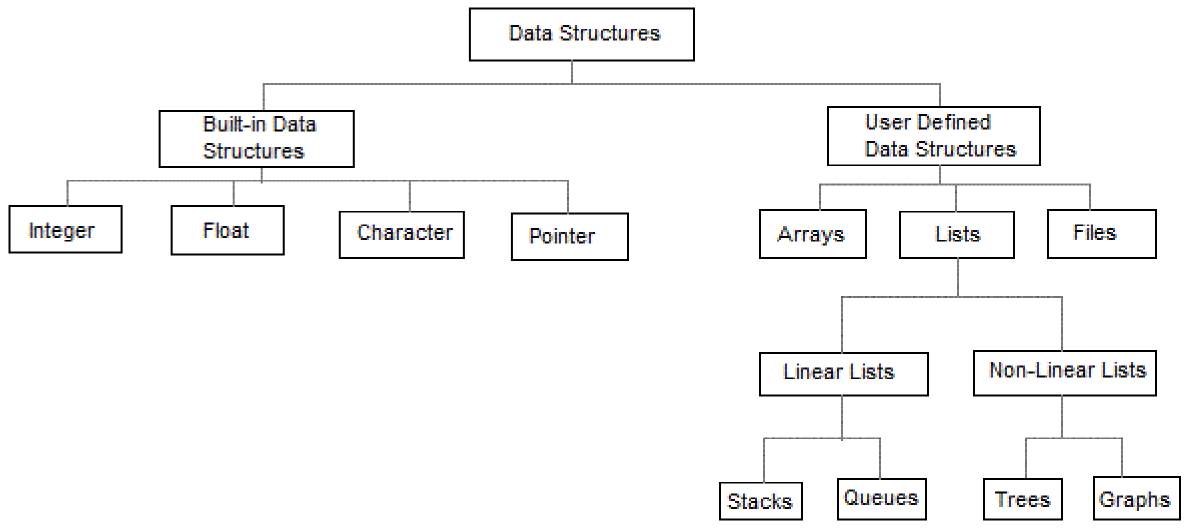
1. Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations. In comparison, a data structure is a concrete implementation of the specification provided by an ADT.
2. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B- tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.
3. Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services.
4. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.
5. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

1.4Types of Data Structures

anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures. Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

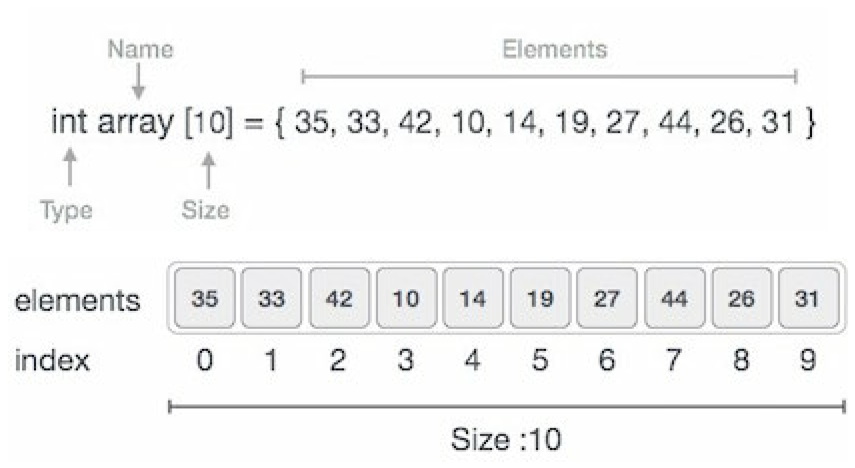
Characterstic	Description
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: Array
Non- Linear	In Non- Linear data structures,the data items are not in sequence. Example: Tree, Graph
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: Array
Non- Homogeneous	In Non- Homogeneous data structure, the elements may or may not be of the same type. Example: Structures
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: Array
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: Linked List created using pointers

1.4.1 Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. 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.

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

1.4.2. Linked List

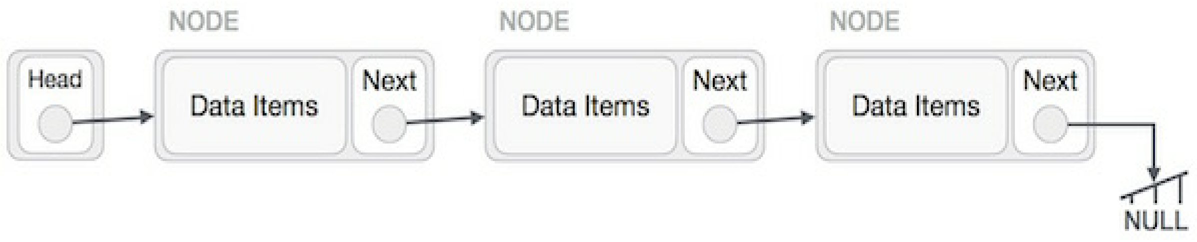
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most- used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.

- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked list can be visualized as a chain of nodes, where every node points to the next node.



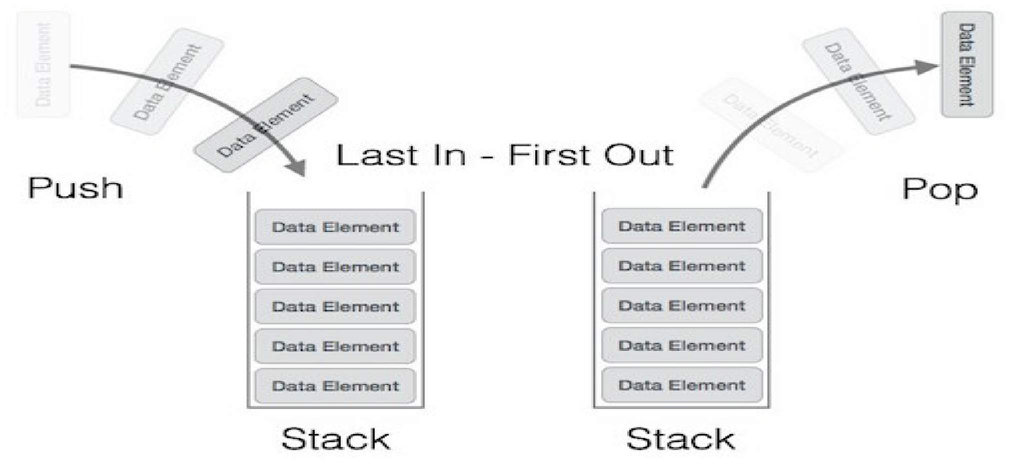
As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

1.4.3. Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real- world stack, for example – a deck of cards or a pile of plates, etc.

A real- world stack allows operations at one end only. 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. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

Basic Operations

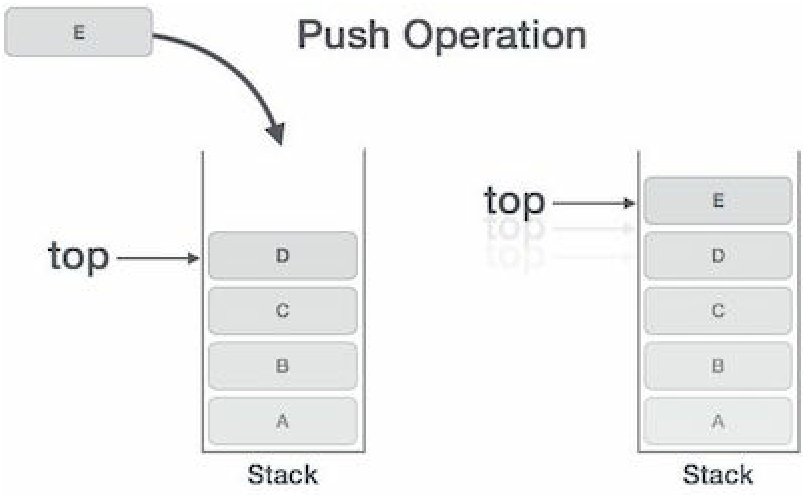
Stack operations may involve initializing the stack, using it and then de- initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

1.Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- Step 1 − Checks if the stack is full.
- Step 2 − If the stack is full, produces an error and exit.
- Step 3 − If the stack is not full, increments top to point next empty space.
- Step 4 − Adds data element to the stack location, where top is pointing.
- Step 5 − Returns success.

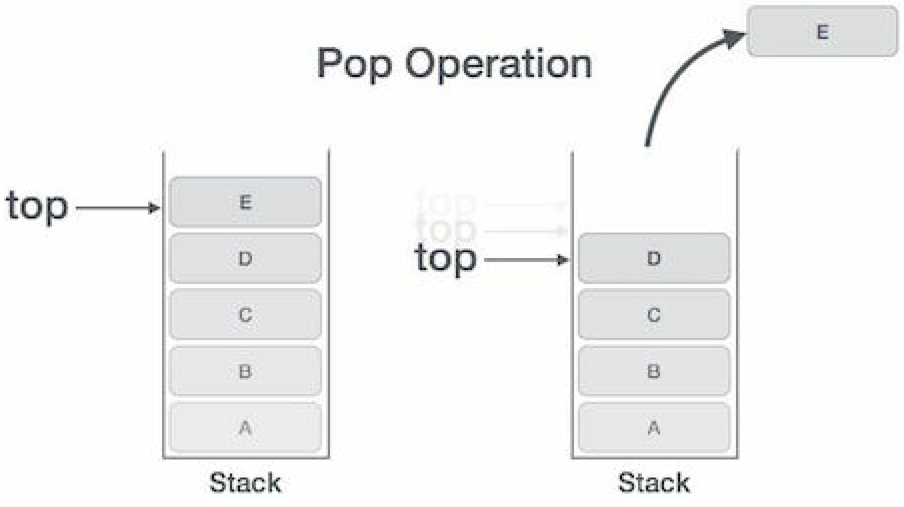


2.Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked- list implementation, pop() actually removes data element and deallocates memory space.

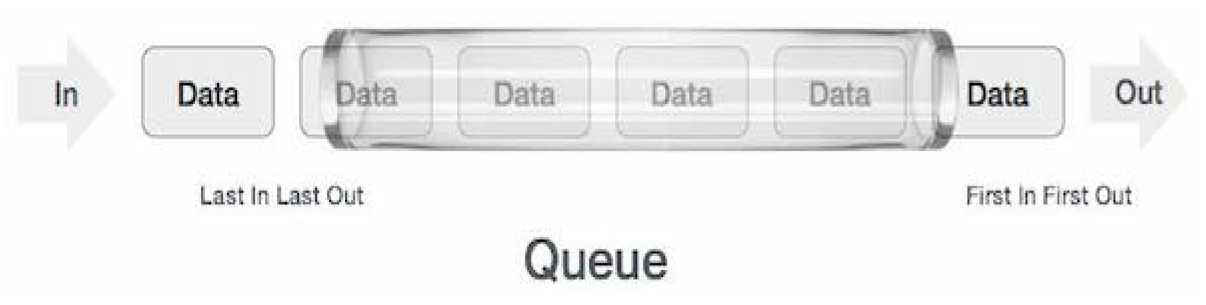
A Pop operation may involve the following steps –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.



1.4.4 Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First- In- First- Out methodology, i.e., the data item stored first will be accessed first.



As in stacks, a queue can also be implemented using Arrays, Linked- lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

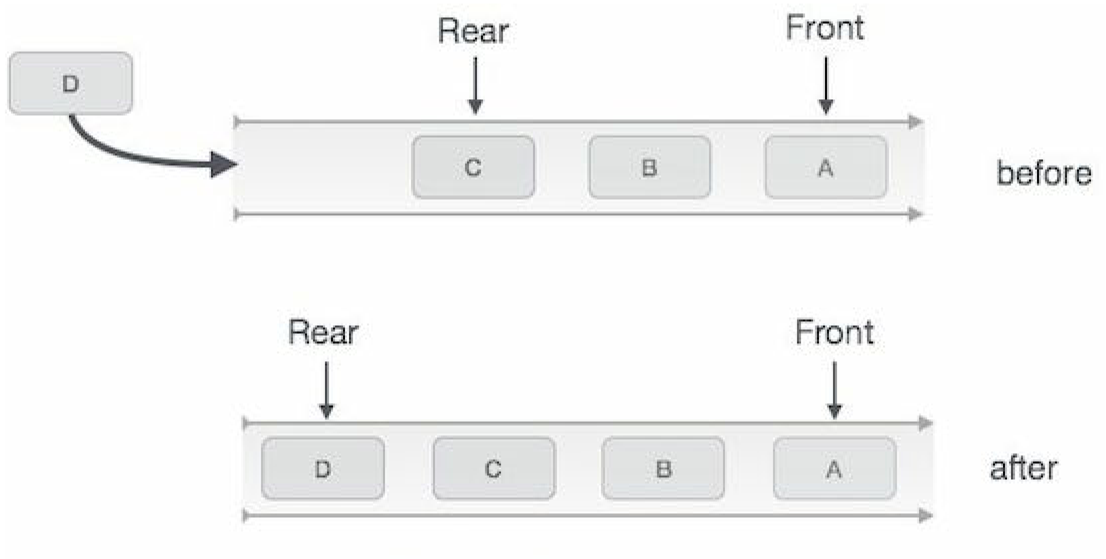
- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

1.Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.

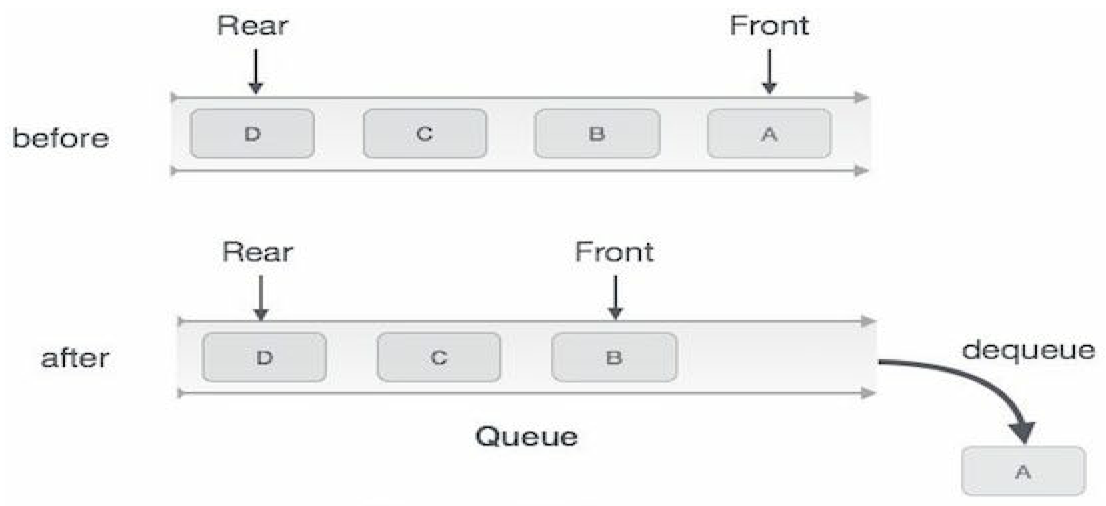


Queue Enqueue

2.Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation –

- Step 1 – Check if the queue is empty.
- Step 2 – If the queue is empty, produce underflow error and exit.
- Step 3 – If the queue is not empty, access the data where front is pointing.
- Step 4 – Increment front pointer to point to the next available data element.
- Step 5 – Return success.

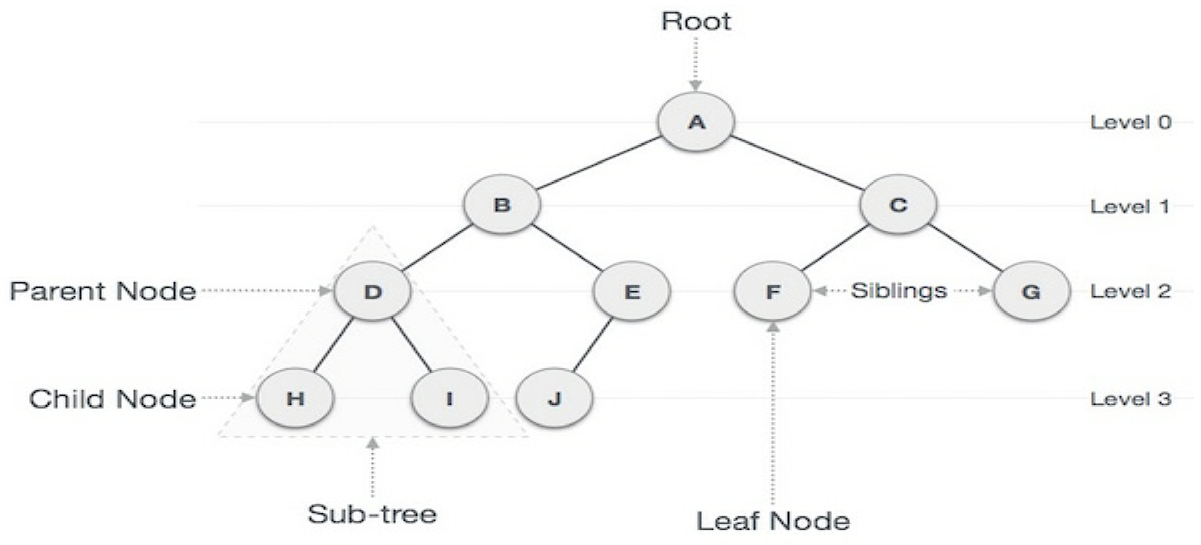


Queue Dequeue

1.4.5Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Following are the important terms with respect to tree.

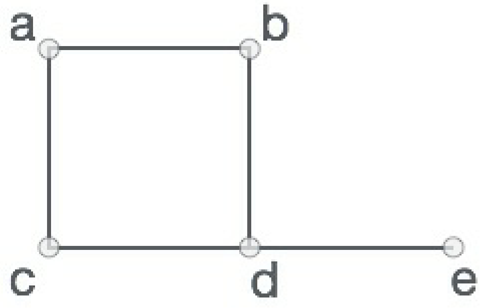
1) **Path** – Path refers to the sequence of nodes along the edges of a tree.
2) **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
3) **Parent** – Any node except the root node has one edge upward to a node called parent.
4) C**hild** – The node below a given node connected by its edge downward is called its child node.
5) **Leaf** – The node which does not have any child node is called the leaf node.
6) **Subtree** – Subtree represents the descendants of a node.
7) **Visiting** – Visiting refers to checking the value of a node when control is on the node.
8) T**raversing** – Traversing means passing through nodes in a specific order.
9) **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

10) **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

1.4.6Graph

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and Eis the set of edges, connecting the pairs of vertices. Take a look at the following graph –



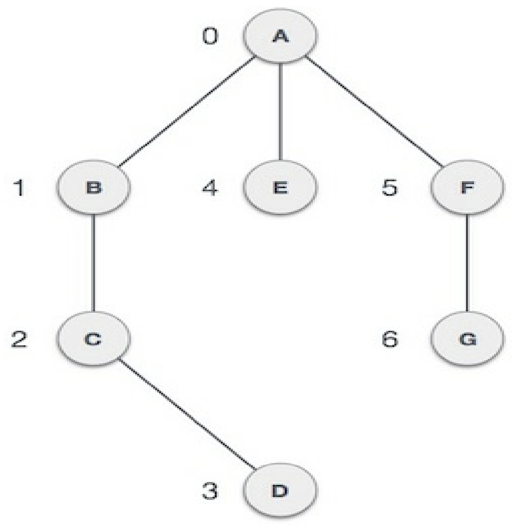
In the above graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two- dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two- dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as

0.

- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph –

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.