

# LECTURE MATERIAL FOR DATA STRUCTURE

*Developed by Engr. Gentle Inyang*

*Data structure* is a systematic way of storing and organizing data in a computer so that it can be used efficiently. A *data structure* is a special format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

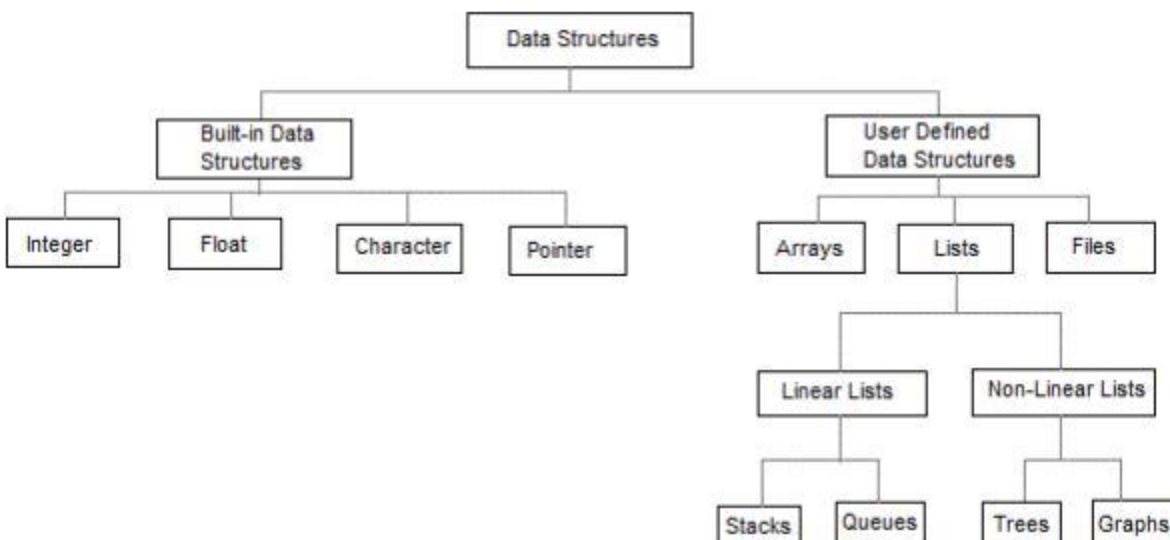
Data Structure is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type. We can organize this data as a record like Player record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

Basic types of Data Structures As we have discussed above, 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

## DIAGRAM REPRESENTING TYPES OF DATA STRUCTURE



## Characteristics of a Data Structure

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

## Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** – Consider an inventory of 1 million (106 ) items of a store. If the application is to search an item, it has to search an item in 1 million (106 ) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** – Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** – As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

## Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- **Worst Case** – This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is  $f(n)$  then this operation will not take more than  $f(n)$  time where  $f(n)$  represents function of  $n$ .
- **Average Case** – This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes  $f(n)$  time in execution, then  $m$  operations will take  $mf(n)$  time.
- **Best Case** – This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes  $f(n)$  time in execution, then the actual operation may take time as the random number which would be maximum as  $f(n)$ .

## BASIC TERMINOLOGY

**Data:** Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

**Group Items:** Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

**Record:** Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

**Files:** A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

**Attribute and Entity:** An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:** Field is a single elementary unit of information representing the attribute of an entity.

**Character:** A character is a display unit of information in computer science that is equivalent to one alphabetic symbol or letter. It includes letters, digits and special symbols such as + (Plus sign), \_ (minus sign), \, /, \$, a, b, ..., z, A, B, ..., Z etc. Every character requires one **byte** of memory unit for storage in computer system.

**Relations:** Relation is sometimes used to refer to a table in a relational database but is more commonly used to describe the relationships that can be created between those tables in a relational database.

**Graph:** A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

**Symbols:** Symbols are diagrammatical representation of data process in an algorithm known as a flowchart.

**Sub Fields:** The date for instance is represented by the day, the month and number is called an elementary item, because it can not be sub-divided into sub-items otherwise known as sub fields called.

## DATA STRUCTURE IN RESPECT TO ATTRIBUTE

The logical or mathematical model of a particular organization of data is called its data structures. A data item is a single unit of values. It is a raw fact which becomes information after processing. Data items for example, date are called group items if they can be divided into subsystems. The date for instance is represented by the day, the month and number is called an elementary item, because it can not be sub-divided into sub-items. It is indeed treated as a single item. An entity is used to describe anything that has certain attributes or properties, which may be assigned values. For example, the following are possible attributes and their corresponding values for an entity known as **STUDENT**.

ATTRIBUTES	NAME	AGE	SEX	MATRIC NO
VALUES	Paul	21	Male	800654

Entities with similar attributes for example, all the 200 level Computer Engineering & Mass communication students form an entity set.

### Value range

All possible values that could be assigned to a given attribute of an entity set is called the range of values of the attribute.

## DATA LIFE CYCLE

The data life cycle, also called the information life cycle, refers to the entire period of time that data exists in your system. This life cycle encompasses all the stages that your data goes through, from first capture onward.

In life science, every living thing undergoes a series of phases: infancy, a period of growth and development, productive adulthood, and old age. These phases vary across the tree of life.

In the same way, different data objects will go through different stages of life at their own cadences. That said, here's one example of a data lifecycle framework:

1. Data creation, ingestion, or capture

- Whether you generate data from data entry, acquire existing data from other sources, or receive signals from devices, you get information somehow. This stage describes when data values enter the firewalls of your system.

2. Data processing

- There are many processes involved in cleaning and preparing raw data for later analysis. While the order of operations may vary, data preparation typically includes integrating data from multiple sources, validating data, and applying the transformation. Data is often reformatted, summarized, subset, standardized, and enriched as part of the data processing workflow.

3. Data analysis

- However you analyze and interpret your data, this is where the magic happens. Exploring and interpreting your data may require a variety of analyses. This could mean statistical analysis and visualization. It can also mean using traditional data modeling or applying artificial intelligence (AI).

4. Data sharing or publication

- This stage is where forecasts and insights turn into decisions and direction. When you disseminate the information gained from data analysis, your data delivers its full business value.

5. Archiving

- Once data has been collected, processed, analyzed, and shared, it is typically stored for future reference. For archives to have any future value, it's important to keep metadata about each item in your records, particularly about data provenance.

## **Big Data Life Cycle**

It's not news to anyone that the volumes of data have grown enormously in recent years, and are only continuing to grow. You don't necessarily need (or want) to collect all the data in the universe. While it might sound nice to have every scrap of information at your disposal, data management challenges scale with data volume. More data means higher data storage costs. The more data you have, the more resources you'll need for data preparation and analysis. In order to scale up for big data without going overboard, build some precautions into your big data life cycle. Three activities typically lead to problems: over-collecting data, managing it poorly, and hoarding deprecated data. Here's what to do instead:

1. Refine your data collection process

- To avoid data overcollection, don't collect all generated data. Instead, create a plan to define and capture only the data that's relevant to your project.

## 2. Implement effective data management

- Catalog your data so it's easy to find and use, and create an infrastructure that combines manual and automated monitoring and maintenance to maintain the health of that data.

## 3. Dispose of unnecessary data

- Once they have outlived their usefulness, consider deleting data or purging old records. You'll want to keep in mind any legal obligations to either maintain or delete old records and establish a clear schedule for data deletion.

## ALGORITHM

A finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time. Whatever the input values, an algorithm will definitely terminate after executing a finite number of instructions.

### Characteristics of algorithm:

- Has a finite set of steps with definite instructions.
- Instructions have definite order.
- Algorithm must eventually stop.
- Actions are deterministic.

### How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm. We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

### Example

Let's try to learn algorithm-writing by using an example.

**Problem** – Design an algorithm to add two numbers and display the result.

**step 1** – START

**step 2** – declare three integers **a**, **b** & **c**

**step 3** – define values of **a** & **b**

**step 4** – add values of **a** & **b**

**step 5** – store output of step 4 to **c**

**step 6** – print c

**step 7** – STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

**step 1** – START ADD

**step 2** – get values of a & b

**step 3** –  $c \leftarrow a + b$

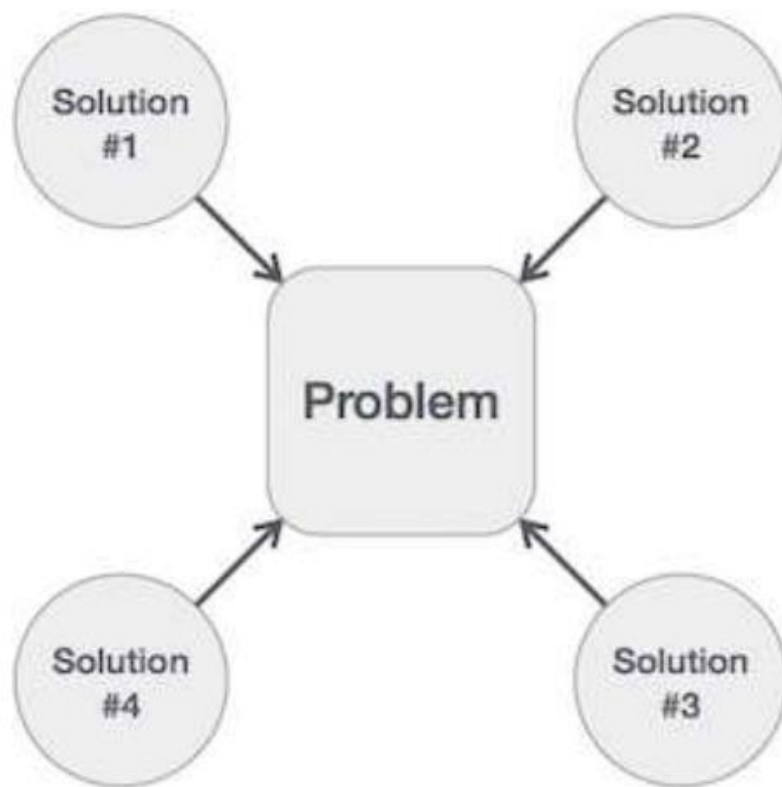
**step 4** – display c

**step 5** – STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

## Data Types

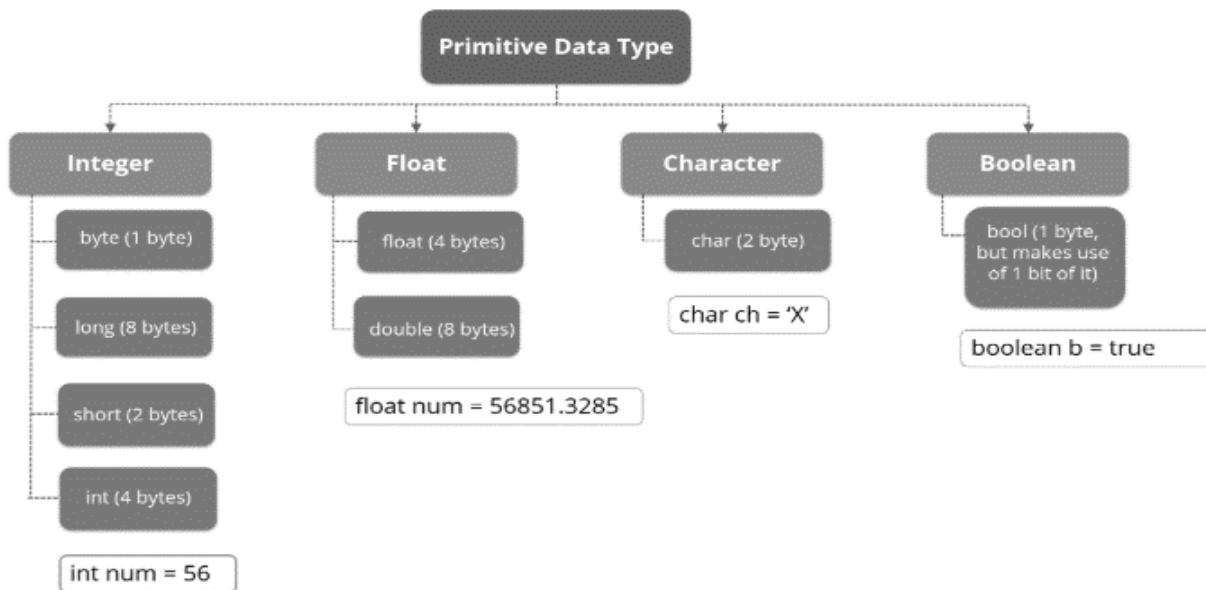
In mathematics it is customary to classify variables according to certain important characteristics. Clear distinctions are made between real, complex, and logical variables or between variables representing individual values, or sets of values, or sets of sets, or between functions, functionals, sets of functions, and so on. This notion of classification is equally if not more important in data processing. We will adhere to the principle that every constant, variable, expression, or function is of a certain *type*. This type essentially characterizes the set of values to which a constant belongs, or which can be assumed by a variable or expression, or which can be generated by a function .

Therefore, a data type is a set of values together with the operations defined on the values: {(values) (operations)}. The operations are performed on the values defined. E.g integer (-4,-1,1,3,4) are values while(+,-,\*,/) are operations. Data types also allow us to associate meaning to sequence of bits in the computer memory. Eg string AA@, integer 5 etc .

Data types	Operations	Storage Representation
1. Integer	*,+,-,/	2's complement, sign magnitude
2. Real	""""	""""
3. Boolean	AND, OR, NOT	True=0, False=1
4. Character		8 bits ASCII/EBCDIC
5. String	Concatenation.	length followed by sequence of characters Length, substring, eg for AABC@ we can have: /3/A/B/C/
Pattern matching	sequence of characters terminated by special symbols.	
6. Pointers Value of	"----->" As for integers	

### DATA TYPES can be classified as :

Primitive (Also called built in) eg Integers, real, pointers, Boolean. They are native or local to the language. The language knows their structure. They are part of the original design .



## Standard Primitive Types

Standard primitive types are those types that are available on most computers as built-in features. They include the whole numbers, the logical truth values, and a set of printable characters. On many computers fractional numbers are also incorporated, together with the standard arithmetic operations. We denote these types by the identifiers INTEGER, REAL, BOOLEAN, CHAR.

### Integer types

The type INTEGER comprises a subset of the whole numbers whose size may vary among individual computer systems.

### The type REAL

The type REAL denotes a subset of the real numbers. Whereas arithmetic with operands of the types INTEGER is assumed to yield exact results, arithmetic on values of type REAL is permitted to be inaccurate within the limits of round-off errors caused by computation on a finite number of digits. This is the principal reason for the explicit distinction between the types INTEGER and REAL, as it is made in most programming languages. The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (\*), and division (/). It is an essence of data typing that different types are incompatible under assignment. An exception to this rule is made for assignment of integer values to real variables, because here the semantics are unambiguous. After all, integers form a subset of real numbers.

### The type CHAR

The standard type CHAR comprises a set of printable characters. Unfortunately, there is no generally accepted standard character set used on all computer systems. Therefore, the use of the predicate "standard" may in this case be almost misleading; it is to be understood in the sense of "standard on the computer system on which a certain program is to be executed." The character set defined by the International Standards Organization (ISO), and particularly its American version ASCII (American Standard Code for Information Interchange) is the most widely accepted set. It consists of 95 printable (graphic) characters and 33 control characters, the latter mainly being used in data transmission and for the control of printing equipment. In order to be able to design algorithms

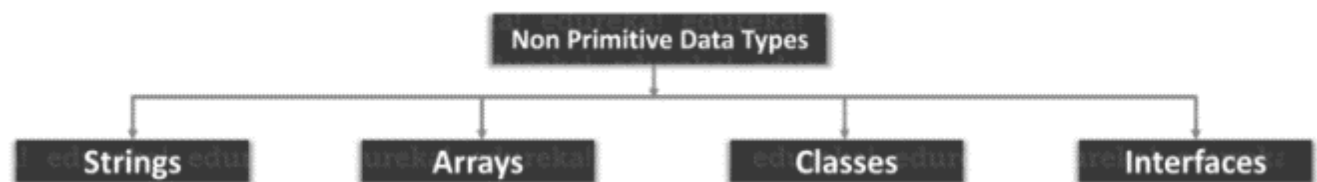


involving characters (i.e., values of type CHAR) that are system independent, we should like to be able to assume certain minimal properties of character sets, namely:

1. The type CHAR contains the 26 capital Latin letters, the 26 lower-case letters, the 10 decimal digits, and a number of other graphic characters, such as punctuation marks.
2. The subsets of letters and digits are ordered and contiguous.
3. The type CHAR contains a non-printing, blank character and a line-end character that may be used as separators.

### Non Primitive Data Structure (User Defined Structure)

These data types are not actually defined by the programming language but are created by the programmer. They are also called “reference variables” or “object references” since they reference a memory location which stores the data.



Example of non primitive data types in *java programming language*

## SETS AND RELATIONS

The concept of a set in the mathematical sense has wide application in computer science. The notations and techniques of set theory are commonly used when describing and implementing algorithms because the abstractions associated with sets often help to clarify and simplify algorithm design.

A **set** is a collection of distinguishable members or elements. The members are typically drawn from some larger population known as the base type [*The data type for the elements in a set. For example, the set might consist of the integer values 3, 5, and 7. In this example, the base type is integers.*]. Each member of a set is either a primitive element of the base type or is a set itself. There is no concept of duplication in a set. Each value from the base type is either in the set or not in the set. For example, a set named **P** might consist of the three integers 7, 11, and 42. In this case, P's members are 7, 11, and 42, and the base type is integer.

An element of a set includes the below example

1. {1, 3, 9, 12}
2. {red, orange, yellow, green, blue, indigo, purple}

The following table shows the symbols commonly used to express sets and their relationships.

$\{1, 4\}$	A set composed of the members 1 and 4
$\{x \mid x \text{ is a positive integer}\}$	A set definition using a set former Example: the set of all positive integers
$x \in P$	$x$ is a member of set $P$
$x \notin P$	$x$ is not a member of set $P$
$\emptyset$	The null or empty set
$ P $	Cardinality: size of set $P$ or number of members for set $P$
$P \subseteq Q, Q \supseteq P$	Set $P$ is included in set $Q$ , set $P$ is a subset of set $Q$ , set $Q$ is a superset of set $P$
$P \cup Q$	Set Union: all elements appearing in $P$ OR $Q$
$P \cap Q$	Set Intersection: all elements appearing in $P$ AND $Q$
$P - Q$	Set difference: all elements of set $P$ NOT in set $Q$
$P \times Q$	Set (Cartesian) Product: yields a set of ordered pairs

Here are some examples of this notation in use. First define two sets,  $P$  and  $Q$ .

$$P = \{2, 3, 5\}, \quad Q = \{5, 10\}.$$

$|P| = 3$  (because  $P$  has three members) and  $|Q| = 2$  (because  $Q$  has two members). Both of these sets are finite in length. Other sets can be infinite, for example, the set of integers.

The union of  $P$  and  $Q$ , written  $P \cup Q$ , is the set of elements in either  $P$  or  $Q$ , which is  $\{2, 3, 5, 10\}$ . The intersection of  $P$  and  $Q$ , written  $P \cap Q$ , is the set of elements that appear in both  $P$  and  $Q$ , which is  $\{5\}$ . The set difference of  $P$  and  $Q$ , written  $P - Q$ , is the set of elements that occur in  $P$  but not in  $Q$ , which is  $\{2, 3\}$ . Note that  $P \cup Q = Q \cup P$  and that  $P \cap Q = Q \cap P$ , but in general  $P - Q \neq Q - P$ . In this example,  $Q - P = \{10\}$ . Finally, the set  $\{5, 3, 2\}$  is indistinguishable from set  $P$ , because sets have no concept of order. Likewise, set  $\{2, 3, 2, 5\}$  is also indistinguishable from  $P$ , because sets have no concept of duplicate elements.

The set product or Cartesian product of two sets  $Q \times P$  is a set of ordered pairs. For our example sets, the set product would be,

$$\{(2,5), (2,10), (3,5), (3,10), (5,5), (5,10)\}.$$

## SUBSET

Set  $A$  is said to be a subset of Set  $B$  if all the elements of Set  $A$  are also present in Set  $B$ . In other words, set  $A$  is contained inside Set  $B$ .

Example: If set  $A$  has  $\{X, Y\}$  and set  $B$  has  $\{X, Y, Z\}$ , then  $A$  is the subset of  $B$  because elements of  $A$  are also present in set  $B$ .

In set theory, a subset is denoted by the symbol  $\subseteq$  and read as 'is a subset of'. Using this symbol we can express subsets as follows:

$A \subseteq B$ ; which means Set A is a subset of Set B.

The subsets of any set consists of all possible sets including its elements and the null set. Let us understand with the help of an example.

**Example:** Find all the subsets of set  $A = \{1,2,3,4\}$

Solution: Given,  $A = \{1,2,3,4\}$

Subsets =

$\{\}$

$\{1\}, \{2\}, \{3\}, \{4\},$

$\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}, \{3,4\},$

$\{1,2,3\}, \{2,3,4\}, \{1,3,4\}, \{1,2,4\}$

$\{1,2,3,4\}$

### Types of Subsets

Subsets are classified as

- Proper Subset
- Improper Subsets

A proper subset is one that contains a few elements of the original set whereas an improper subset, contains every element of the original set along with the null set.

**For example,** if set  $A = \{2, 4, 6\}$ , then,

Number of subsets:  $\{2\}, \{4\}, \{6\}, \{2,4\}, \{4,6\}, \{2,6\}, \{2,4,6\}$  and  $\Phi$  or  $\{\}$ .

Proper Subsets:  $\{\}, \{2\}, \{4\}, \{6\}, \{2,4\}, \{4,6\}, \{2,6\}$

Improper Subset:  $\{2,4,6\}$

There is no particular formula to find the subsets, instead, we have to list them all, to differentiate between proper and improper one.

Proper Subset Symbol

A proper subset is denoted by  $\subset$  and is read as 'is a proper subset of'. Using this symbol, we can express a proper subset for set A and set B as;

$$A \subset B$$

### How many subsets and proper subsets does a set have?

If a set has "n" elements, then the number of subset of the given set is  $2^n$  and the number of proper subsets of the given subset is given by  $2^n - 1$ .

Consider an example, If set A has the elements,  $A = \{a, b\}$ , then the proper subset of the given subset are  $\{\}$ ,  $\{a\}$ , and  $\{b\}$ .

Here, the number of elements in the set is 2.

We know that the formula to calculate the number of proper subsets is  $2^n - 1$ .

$$= 2^2 - 1$$

$$= 4 - 1$$

$$= 3$$

Thus, the number of proper subset for the given set is 3 ( $\{\}$ ,  $\{a\}$ ,  $\{b\}$ ).

### **Improper Subset**

A subset which contains all the elements of the original set is called an improper subset. It is denoted by  $\subseteq$ .

**For example:** Set  $P = \{2, 4, 6\}$  Then, the subsets of P are;

$\{\}$ ,  $\{2\}$ ,  $\{4\}$ ,  $\{6\}$ ,  $\{2, 4\}$ ,  $\{4, 6\}$ ,  $\{2, 6\}$  and  $\{2, 4, 6\}$ .

Where,  $\{\}$ ,  $\{2\}$ ,  $\{4\}$ ,  $\{6\}$ ,  $\{2, 4\}$ ,  $\{4, 6\}$ ,  $\{2, 6\}$  are the proper subsets and  $\{2, 4, 6\}$  is the improper subsets. Therefore, we can write  $\{2, 4, 6\} \subseteq P$ .

### **Power Set**

The power set is said to be the collection of all the subsets. It is represented by  $P(A)$ .

If A is set having elements  $\{a, b\}$ . Then the power set of A will be;

$$P(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$$

**Example 1:** Find the number of subsets and the number of proper subsets for the given set  $A = \{5, 6, 7, 8\}$ .

**Solution:**

Given:  $A = \{5, 6, 7, 8\}$

The number of elements in the set is 4

We know that,

The formula to calculate the number of subsets of a given set is  $2^n$

$$= 2^4 = 16$$

Number of subsets is 16

The formula to calculate the number of proper subsets of a given set is  $2^n - 1$

$$= 2^4 - 1$$

$$= 16 - 1 = 15$$

The number of proper subsets is 15.

### Example of proper and improper subsets.

Proper subset:

$$X = \{2, 5, 6\} \text{ and } Y = \{2, 3, 5, 6\}$$

Improper Subset:

$$X = \{A, B, C, D\} \text{ and } Y = \{A, B, C, D\}$$

### Universal Set

A universal set (usually denoted by  $U$ ) is a set which has elements of all the related sets, without any repetition of elements. Say if  $A$  and  $B$  are two sets, such as  $A = \{1, 2, 3\}$  and  $B = \{1, a, b, c\}$ , then the universal set associated with these two sets is given by  $U = \{1, 2, 3, a, b, c\}$ .

**Example:** Let us say, there are three sets named as  $A$ ,  $B$  and  $C$ . The elements of all sets  $A$ ,  $B$  and  $C$  is defined as;

$$A = \{1, 3, 6, 8\}$$

$$B = \{2, 3, 4, 5\}$$

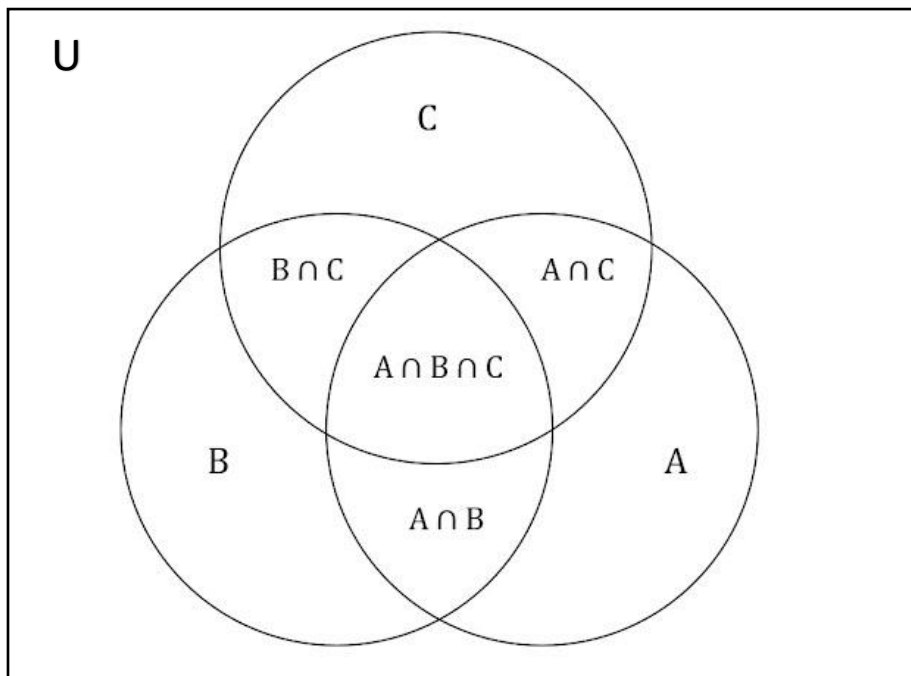
$$C = \{5, 8, 9\}$$

Find the universal set for all the three sets  $A$ ,  $B$  and  $C$ .

**Answer:** By the definition we know, the universal set includes all the elements of the given sets. Therefore, Universal set for sets  $A$ ,  $B$  and  $C$  will be,

$$U = \{1, 2, 3, 4, 5, 6, 8, 9\}$$

### VEIN DIAGRAM REPRESENTATION



## Null Set

A set with no members is called an empty, or null, set, and is denoted  $\emptyset$ .

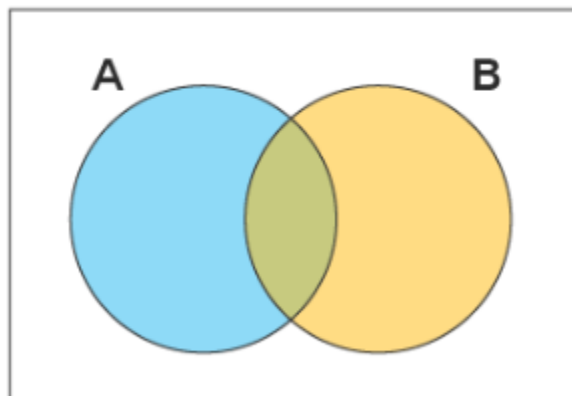
## SET OPERATION

Set operations is a concept similar to fundamental operations on numbers. Sets in math deal with a finite collection of objects, be it numbers, alphabets, or any real-world objects.

There are four main kinds of set operations which are:

- Union of sets
- Intersection of sets
- Complement of a set
- Difference between sets/Relative Complement

Before we move on to discuss the various set operations, let us recall the concept of Venn diagrams as it is important in understanding the operations on sets. A Venn diagram is a logical diagram that shows the possible relationship between different finite sets. It looks as shown below.



### Union of Sets

For two given sets A and B,  $A \cup B$  (read as A union B) is the set of distinct elements that belong to set A and B or both. The number of elements in  $A \cup B$  is given by  $n(A \cup B) = n(A) + n(B) - n(A \cap B)$ , where  $n(X)$  is the number of elements in set X. To understand this set operation of the union of sets better, let us consider an example: If  $A = \{1, 2, 3, 4\}$  and  $B = \{4, 5, 6, 7\}$ , then the union of A and B is given by  $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$ .

### Intersection of Sets

For two given sets A and B,  $A \cap B$  (read as A intersection B) is the set of common elements that belong to set A and B. The number of elements in  $A \cap B$  is given by  $n(A \cap B) = n(A) + n(B) - n(A \cup B)$ , where  $n(X)$  is the number of elements in set X. To understand this set operation of the intersection of sets better, let us consider an example: If  $A = \{1, 2, 3, 4\}$  and  $B = \{3, 4, 5, 7\}$ , then the intersection of A and B is given by  $A \cap B = \{3, 4\}$ .

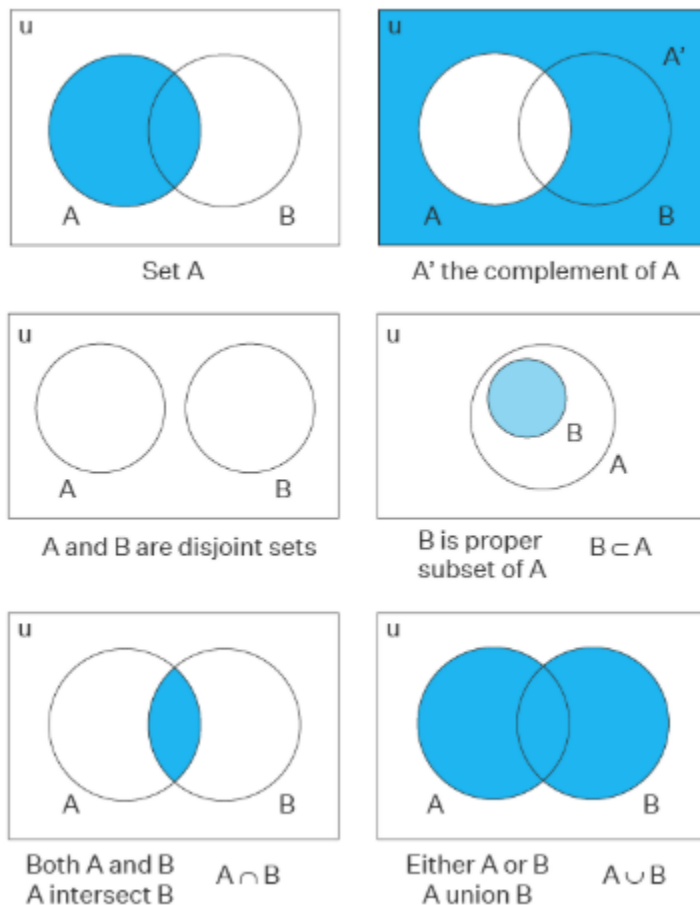
## Complement of Sets

The complement of a set  $A$  denoted as  $A'$  or  $A^c$  (read as  $A$  complement) is defined as the set of all the elements in the given universal set ( $U$ ) that are not present in set  $A$ . To understand this set operation of complement of sets better, let us consider an example: If  $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$  and  $A = \{1, 2, 3, 4\}$ , then the complement of set  $A$  is given by  $A' = \{5, 6, 7, 8, 9\}$ .

## Set Difference

The set operation difference between sets implies subtracting the elements from a set which is similar to the concept of the difference between numbers. The difference between sets  $A$  and  $B$  denoted as  $A - B$  lists all the elements that are in set  $A$  but not in set  $B$ . To understand this set operation of set difference better, let us consider an example: If  $A = \{1, 2, 3, 4\}$  and  $B = \{3, 4, 5, 7\}$ , then the difference between sets  $A$  and  $B$  is given by  $A - B = \{1, 2\}$ .

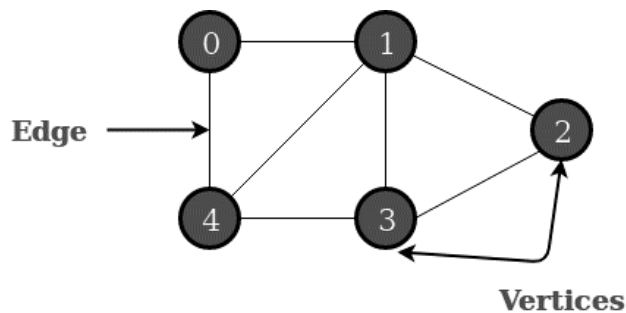
### VEIN DIAGRAM ILLUSTRATING SET OPERATIONS



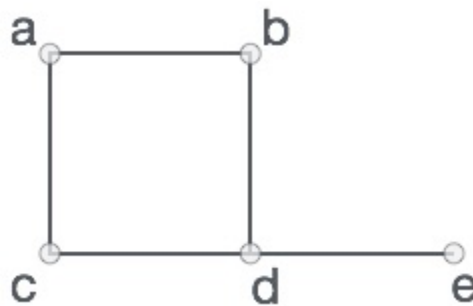
## GRAPHS

A graph is a common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them. A pair  $(x,y)$  is referred to as an edge, which communicates that the  $x$  vertex connects to the  $y$  vertex.

In the examples below, circles represent vertices, while lines represent edges.



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



In the above graph,

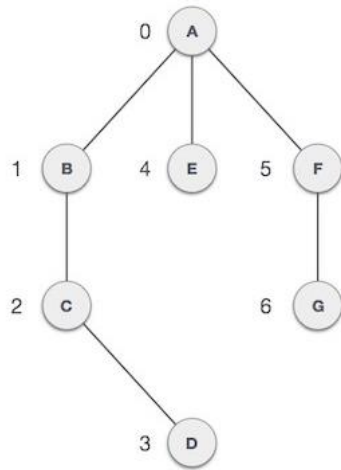
$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

### Properties of Graphs

- **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.



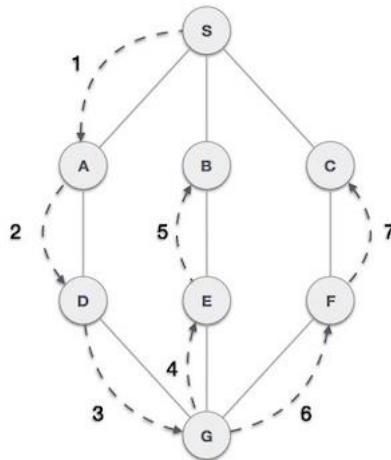


### Basic Operations

Following are basic primary operations of a Graph –

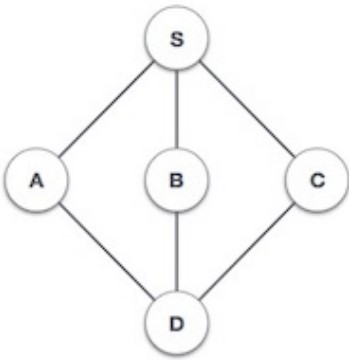

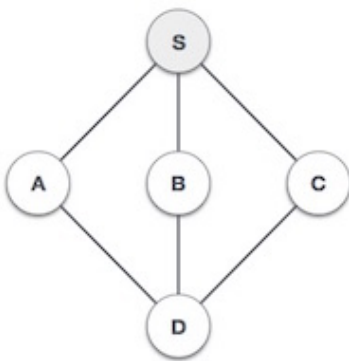
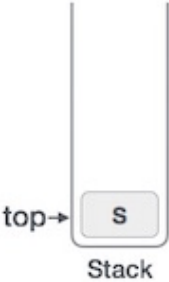
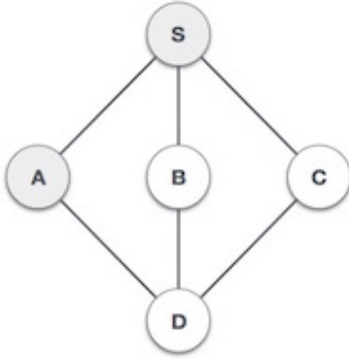
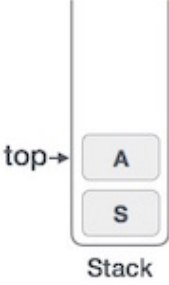
- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

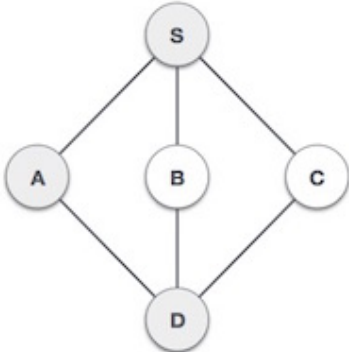

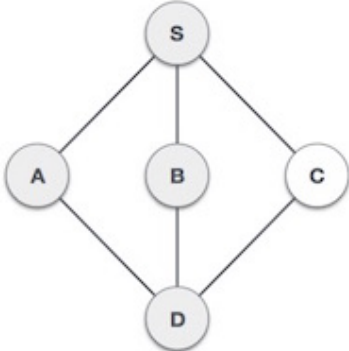
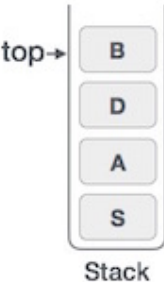
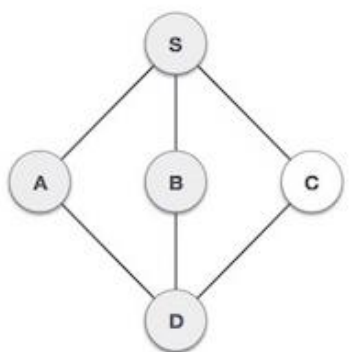

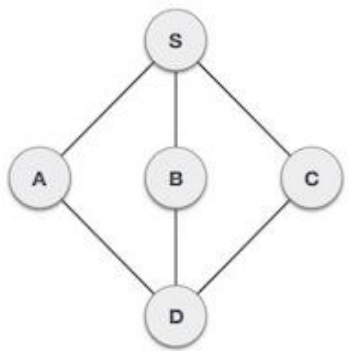
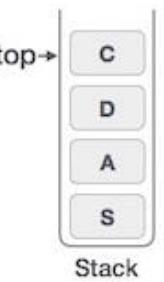
**Depth First Search (DFS)** algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

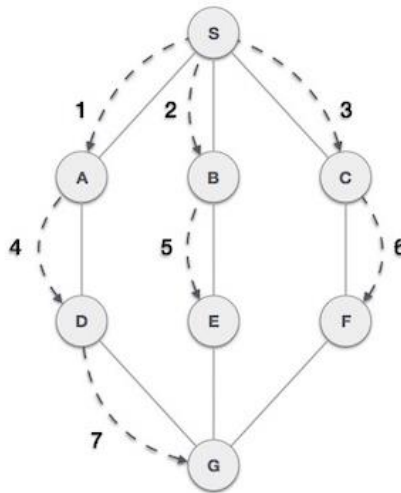
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1	  <p>Stack</p>	Initialize the stack.
2	  <p>Stack</p>	Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3	  <p>Stack</p>	Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.

4	 	<p>Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5	 	<p>We choose <b>B</b>, mark it as visited and put onto the stack. Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</b> from the stack.</p>
6	 	<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of the stack.</p>
7	 	<p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it as visited and put it onto the stack.</p>

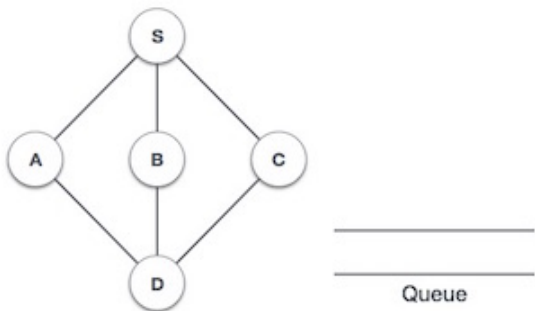
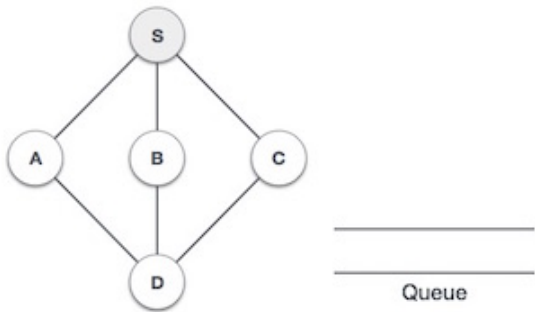
As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

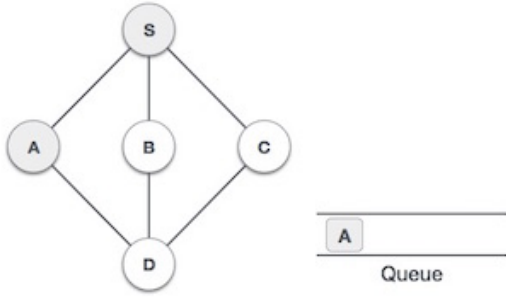
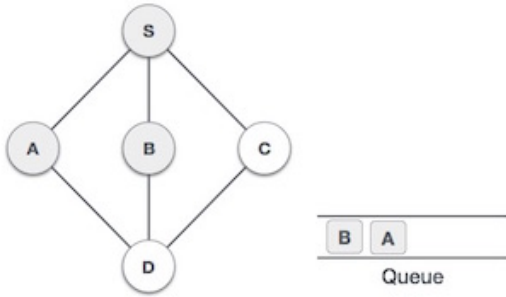
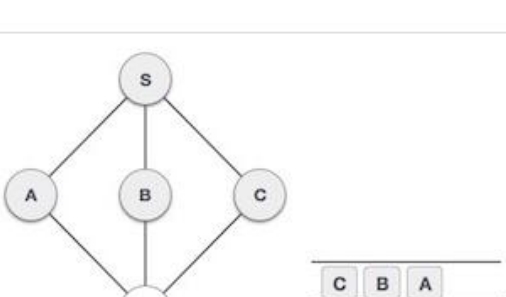
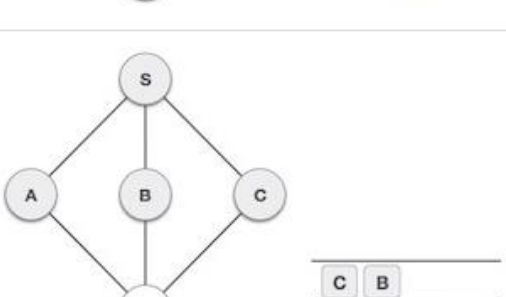
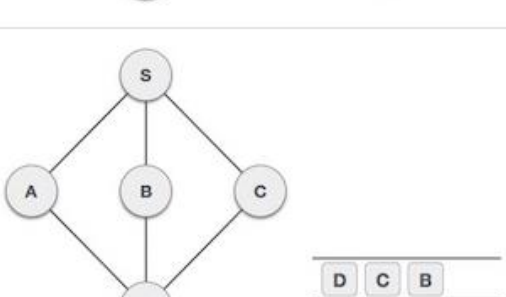
**Breadth First Search (BFS)** algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting <b>S</b> (starting node), and mark it as visited.

3		<p>We then see an unvisited adjacent node from <b>S</b>. In this example, we have three nodes but alphabetically we choose <b>A</b>, mark it as visited and enqueue it.</p>
4		<p>Next, the unvisited adjacent node from <b>S</b> is <b>B</b>. We mark it as visited and enqueue it.</p>
5		<p>Next, the unvisited adjacent node from <b>S</b> is <b>C</b>. We mark it as visited and enqueue it.</p>
6		<p>Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b>.</p>
7		<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.</p>

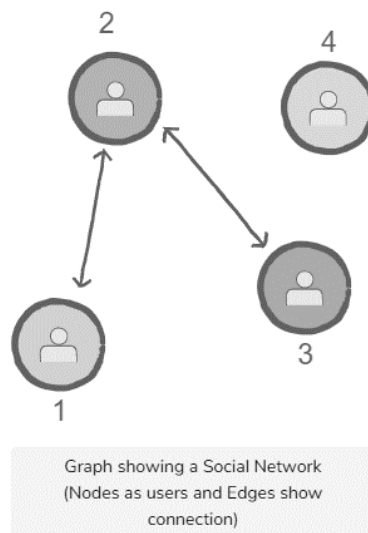
At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Graphs are a commonly used data structure because they can be used to model many real-world problems. A graph consists of a set of nodes with an arbitrary number of connections, or edges, between the nodes. These edges can be either directed or undirected and weighted or unweighted.

Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.).

For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes.

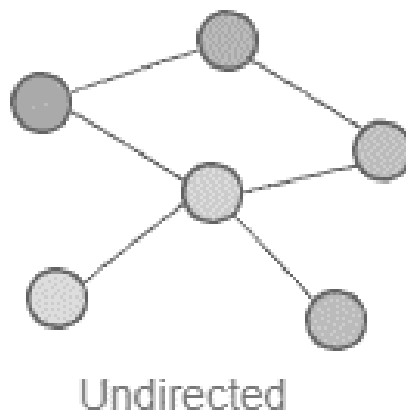
Each node can be a structure that contains information like user's id, name, gender, etc.



### Types of graphs:

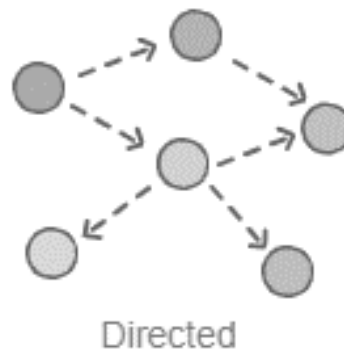
#### Undirected Graph:

In an undirected graph, nodes are connected by edges that are all bidirectional. For example if an edge connects node 1 and 2, we can traverse from node 1 to node 2, and from node 2 to 1.



## Directed Graph

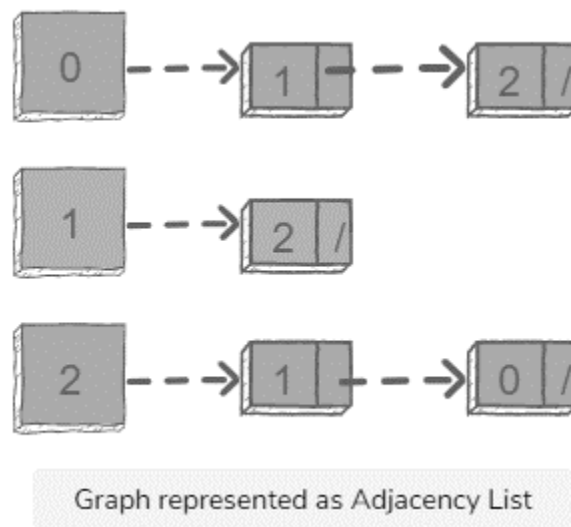
In a directed graph, nodes are connected by directed edges – they only go in one direction. For example, if an edge connects node 1 and 2, but the arrow head points towards 2, we can only traverse from node 1 to node 2 – not in the opposite direction.



## Types of Graph Representations:

### Adjacency List

To create an Adjacency list, an array of lists is used. The size of the array is equal to the number of nodes. A single index,  $\text{array}[i]$  represents the list of nodes adjacent to the  $i$ th node.



### Adjacency Matrix:

An Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of nodes in a graph. A slot  $\text{matrix}[i][j] = 1$  indicates that there is an edge from node  $i$  to node  $j$ .

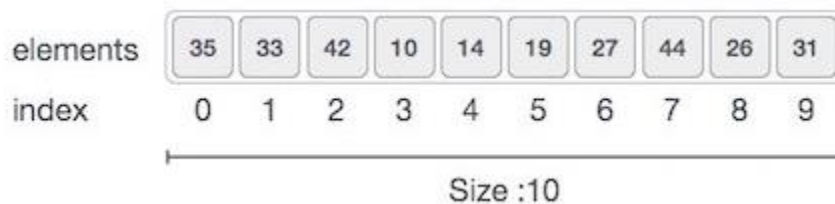
	0	1	2	3
0	1			1
1			1	
2		1		
3	1			1

Graph represented as Adjacency Matrix

**Array:** The simplest type of data structure is a linear array. This is also called one dimensional array. An array holds several values of the same kind. Accessing the elements is very fast. It may not be possible to add more values than defined at the start, without copying all values into a new array. In computer science, an array data structure or simply an array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.

Name
Elements  
↓
↓  
`int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }`  
↑
↑  
Type
Size

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

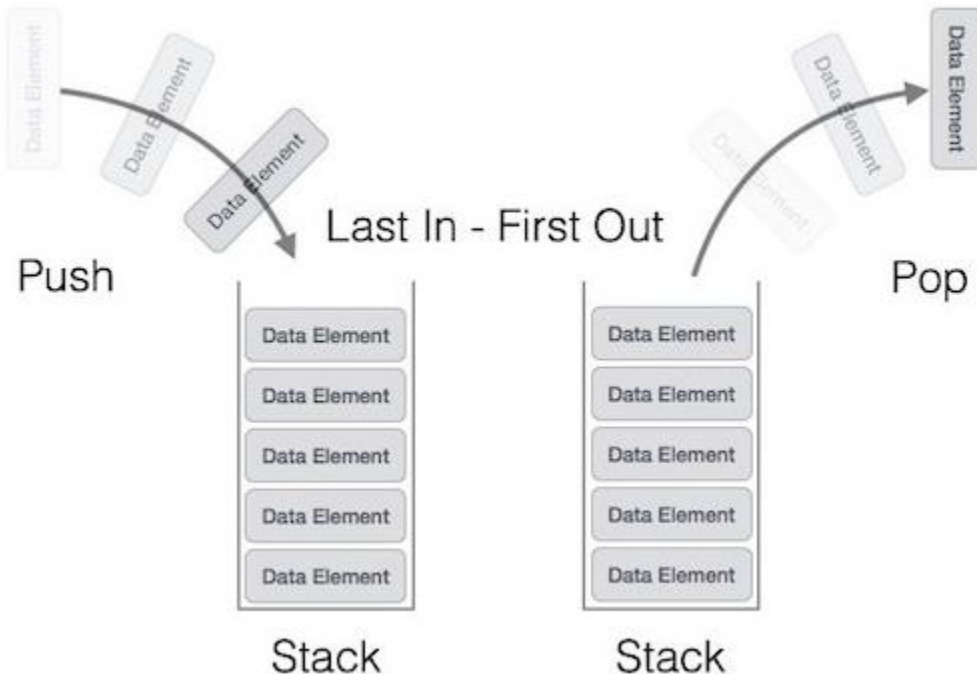


**Stacks:** A stack is an abstract data type that holds an ordered, linear sequence of items. In contrast to a queue, a stack is a last in, first out (LIFO) structure. A real-life example is a stack of plates: you can only take a plate from the top of the stack, and you can only add a plate to the top of the stack. If you want to reach a plate that is not on the top of the stack, you need to remove all of the plates that are above that one. In the same way, in a stack data structure, you can only access the element on the top of the stack. The element that was added last will be the one to be removed first. Therefore, to implement a stack, you need to maintain a pointer to the top of the stack (the last element to be added).



### Stack Representation:

The following diagram depicts a stack and its operations



**Queues:** A queue is an important data structure in programming. A queue follows the FIFO (First In First Out) method and is open at both of its ends. Data insertion is done at one end rear end or the tail of the queue while deletion is done at the other end called the front end or the head of the queue.

A real-life example of a queue data structure is a line of people waiting to buy a ticket at a cinema hall. A new person will join the line from the end and the person standing at the front will be the first to get the ticket and leave the line. Similarly in a queue data structure, data added first will leave the queue first.



**Linked List:** A linked list data structure is a set of records linked together by references. The records are often called nodes. The references are often called links or pointers. From here on, the words node and pointer will be used for these concepts.



Each node points to another node.

In linked data structures, pointers are only dereference or compared for equality. Thus, linked data structures are different than arrays, which require adding and subtracting pointers.

### **Advantages and Disadvantages of Linked List**

It is a data structure in which elements are linked using pointers. A node represents an element in linked list which have some data and a pointer pointing to next node. Its structure looks like as shown in below above.

There are various merits and demerits of linked list that I have shared below.

#### **Advantages of Linked List**

##### **Dynamic Data Structure**

Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give initial size of linked list.

##### **Insertion and Deletion**

Insertion and deletion of nodes are really easier. Unlike array here we don't have to shift elements after insertion or deletion of an element. In linked list we just have to update the address present in next pointer of a node.

##### **No Memory Wastage**

As size of linked list can increase or decrease at run time so there is no memory wastage. In case of array there is lot of memory wastage, like if we declare an array of size 10 and store only 6 elements in it then space of 4 elements are wasted. There is no such problem in linked list as memory is allocated only when required.

##### **Implementation**

Data structures such as stack and queues can be easily implemented using linked list.

#### **Disadvantages of Linked List**

##### **Memory Usage**

More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.

##### **Traversal**

Elements or nodes traversal is difficult in linked list. We can not randomly access any element as we do in array by index. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large.

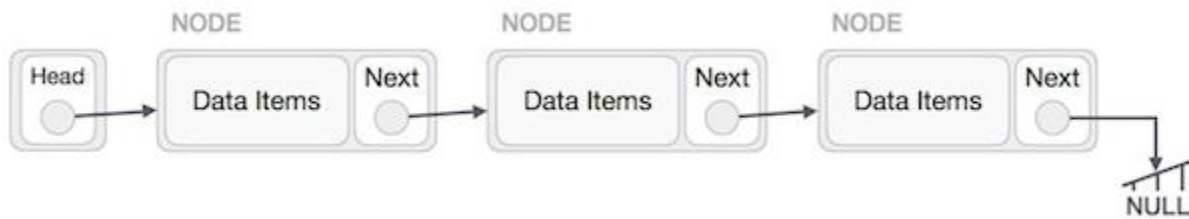
##### **Reverse Traversing**

In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

If you know some other advantages and disadvantages of linked list then please mention by commenting below.

#### **Linked List Representation**

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.

### Types of Linked List

Following are the various types of linked list.

- Simple Linked List – Item navigation is forward only.
- Doubly Linked List – Items can be navigated forward and backward.
- Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

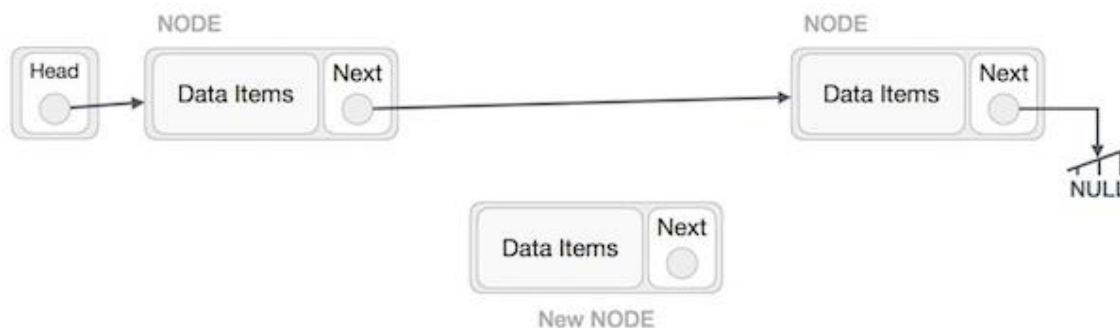
### Basic Operations

Following are the basic operations supported by a list.

- Insertion – Adds an element at the beginning of the list.
- Deletion – Deletes an element at the beginning of the list.
- Display – Displays the complete list.
- Search – Searches an element using the given key.
- Delete – Deletes an element using the given key.

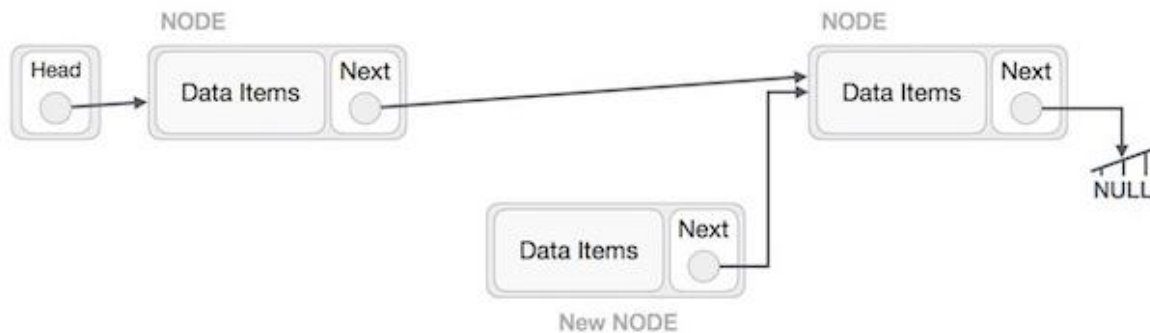
### Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

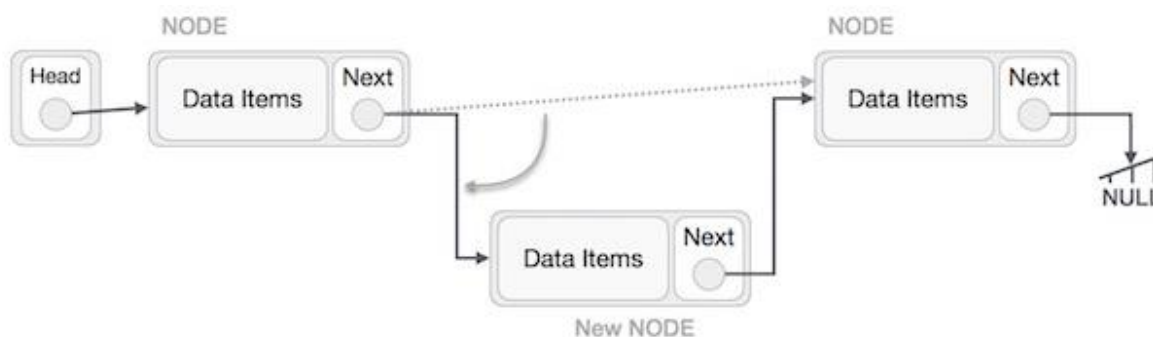


Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B. next to C.

It should look like this:



Now, the next node at the left should point to the new node.



This will put the new node in the middle of the two. The new list should look like this:

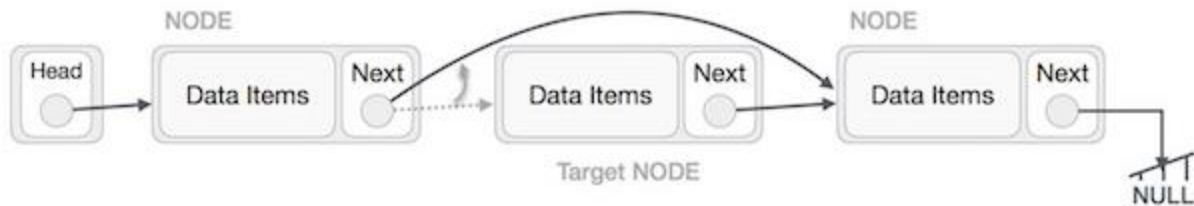


## Deletion Operation

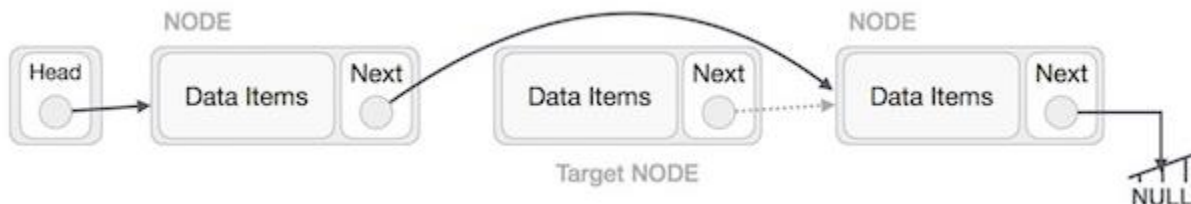
Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



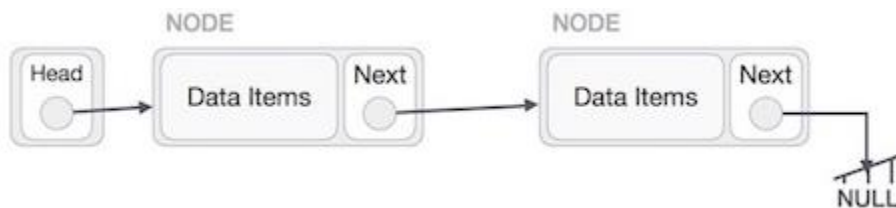
The left (previous) node of the target node now should point to the next node of the target node



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

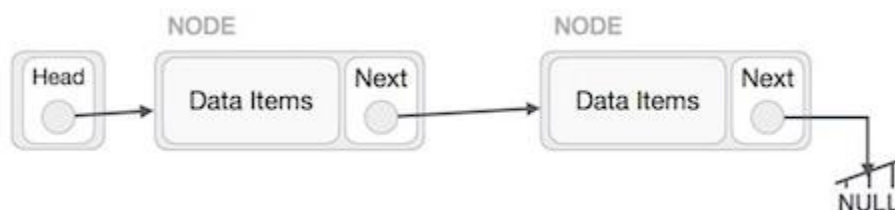


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

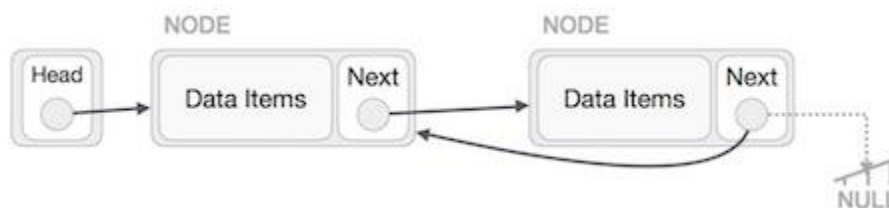


## Reverse Operation

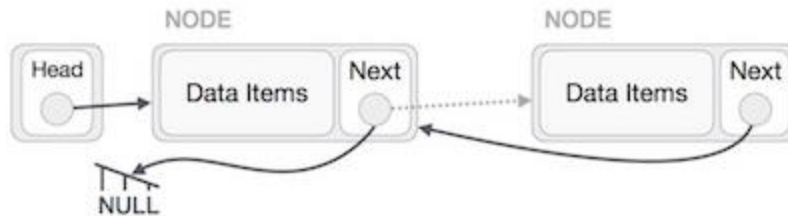
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



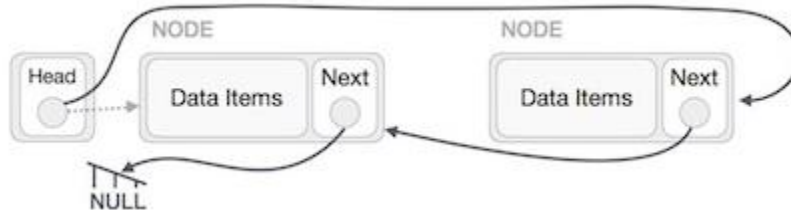
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node



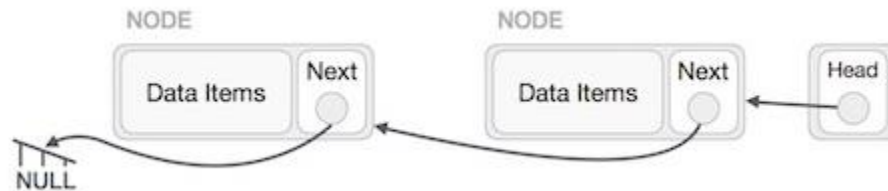
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.

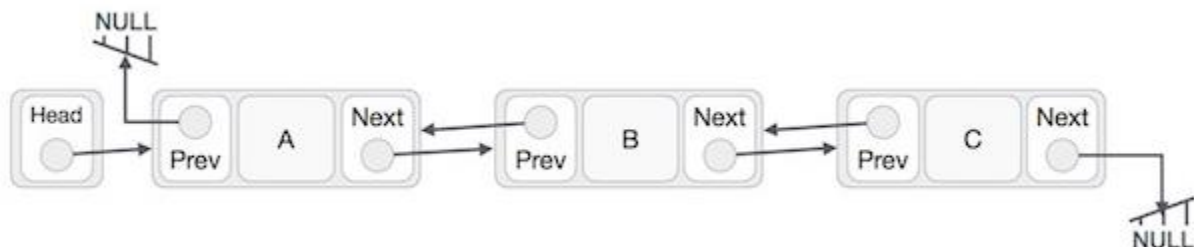


The linked list is now reversed.

**Doubly Linked List** is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly 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.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

### Doubly Linked List Representation



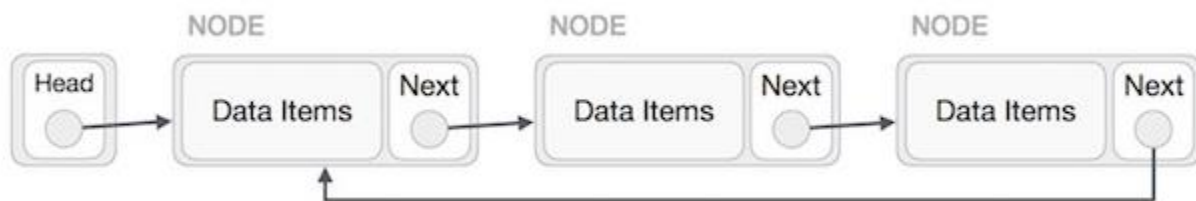
As per the above illustration, following are the important points to be considered.

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

**Circular Linked List** is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

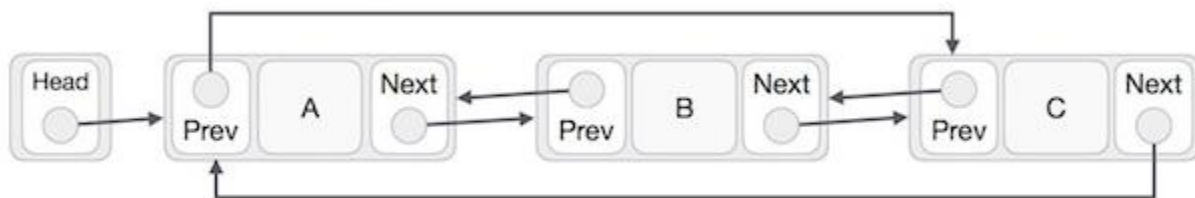
### Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



### Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



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

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

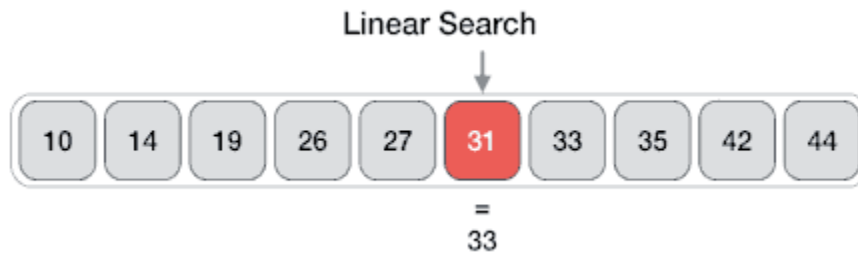
### Basic Operations

Following are the important operations supported by a circular list.

- insert – Inserts an element at the start of the list.
- delete – Deletes an element from the start of the list.
- display – Displays the list.

## SEARCHING AND SORTING TECHNIQUES

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



### Algorithm

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if  $i > n$  then go to step 7

Step 3: if  $A[i] = x$  then go to step 6

Step 4: Set i to  $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Binary search** is a fast search algorithm with run-time complexity of  $O(\log n)$ . This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of  $O(n)$  whereas binary search has  $O(\log n)$ .

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

### How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

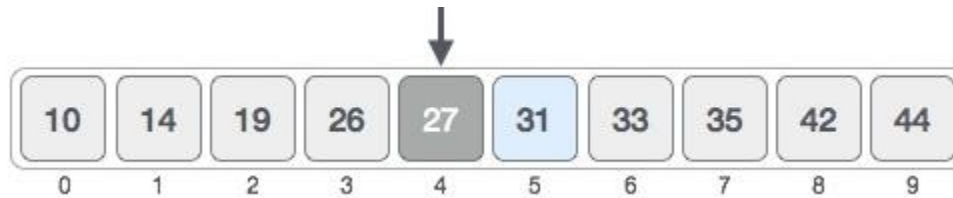




First, we shall determine half of the array by using this formula;

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Here it is,  $0 + (9 - 0) / 2 = 4$  (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

$$\text{low} = \text{mid} + 1$$

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

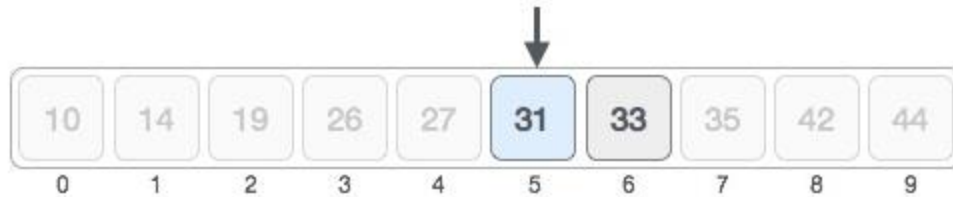
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

## SORTING TECHNIQUES

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where  $n$  is the number of items.

### How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes  $O(n^2)$  time so we're keeping it short and precise.



Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.

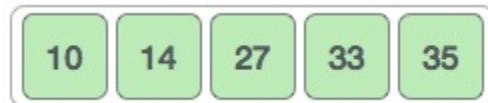


We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this



The process is repeated all over again until the array is sorted correctly hence we have the below result



## Insertion Sorting Technique

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$ , where  $n$  is the number of items.

## How Insertion Sort Works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order, so we swap them to have:



However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

### Selection Sorting Technique

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

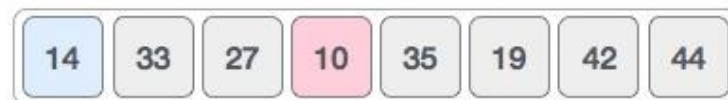
This algorithm is not suitable for large data sets as its average and worst case complexities are of  $O(n^2)$ , where  $n$  is the number of items.

### How Selection Sort Works?

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

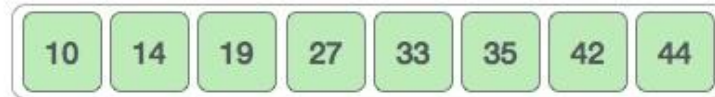
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.



### Merge Sorting Technique

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being  $O(n \log n)$ , it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

### How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this;



Now we should learn some programming aspects of merge sorting.

### Quick Sorting Technique

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are  $O(n^2)$ , respectively.