**What is Data Structures?**

* In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.
* **Data Structures a**re generally **based** on the ability of a computer to fetch (retrieve) and store data at any place in its memory, specified by an address – a bit string that can be itself stored in memory and manipulated by the program.

#### ****How is it implemented?****

Data structure usually requires writing a set of procedures that create and manipulate instances of that structure.

**Classification of Data Structure According to Type**

* **Primitive** – these are basic data structures and are directly operated upon machine instructions, e.g., integer, character.
* **Non-primitive** – these are derived from primitive data structures, e.g., array.

**Classification of Data Structure According to Elements**

* **Homogeneous** – in this data structure, all elements are of the same type, e.g. array.
* **Heterogeneous** – in this data structure, elements are of different types, e.g., structure.

**Classification of Data Structure According to Size**

* **Static** – the size of this data structure cannot be changed after the initial allocation, like matrices.
* **Dynamic** – the size can change dynamically, like in lists.

**Classification of Data Structure According to Relationship**

* **Linear** – this data structure maintains a linear relationship between its elements, e.g., array.
* **Non-linear**– this data structure does not maintain any linear relationship between its elements, e.g., in a tree.

**Memory**

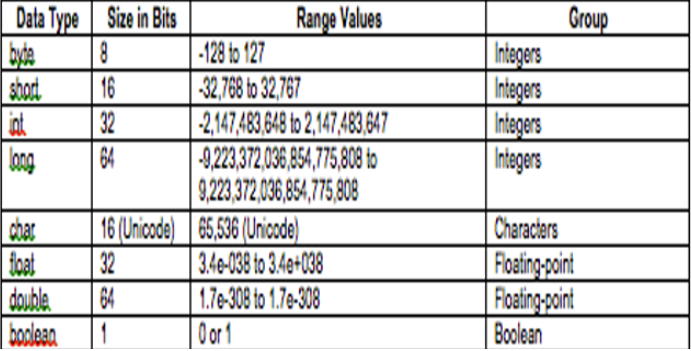
* **Main memory (RAM)** – where instructions (programs) and data are stored; volatile means may be lost once the computer is powered down.
* **Cache memory** in the central processing unit (CPU) – is used to store frequently used instructions and data that either is, will be, or has been used by the CPU. A segment of the CPU’s cache memory is called a register ( a small amount of memory that is used to temporarily store instructions and data.
* **Persistent storage** (external storage devices) – used to store instructions and data in external devices such as a hard disk; non-volatile; commonly used by the operating system as virtual memory (a technique an OS uses to increase the main memory capacity beyond the RAM).
  + Note: *as to access – CPU cache memory is the type of memory that has the fastest access speed, followed by the main memory, and lastly persistent storage because it usually involves a mechanical process that inhibits the quick transfer of instructions and data.*
* **Reserving Memory**
  + Data used by a program is stored in memory and manipulated by various data structure techniques, depending on the nature of the program.
  + Memory is organized into groups of eight bits called a byte, enabling 256 combinations of zeroes and ones that can store numbers from 0 through 255.
  + Data used in a program can be larger than a byte and requires 2, 4, or 8 bytes to be stored in memory. Before any data can be stored in memory, you must tell the computer how much space to reserve for data by using an abstract data type.

#### ****What is an Abstract Data Type(ADT)?****

It is a **keyword** of a programming language that **specifies the amount of memory needed to store data** and **the kind of data that will be stored in that memory location**.

The number of bytes reserved for an ADT varies, depending on the programming language used to write the program and the type of computer used to compile the program.

**Example of Abstract Data Type(ADT)?**

* Java ADT has a fixed size in order for programs to run in all Java runtime environments (JRE).
* C and C++ ADT have sizes based on the register size of the computer used to compile the program. The int and float data types are the size of the register.

**Abstract Data Type Groups**

* Integer
* Floating-point
* Character
* Boolean

**ADT - Integer**

**Integer** – stores whole numbers and signed numbers.

* **byte ADT**(Java reserves 8 bits of main memory)

–It is the smallest ADT in the integer group and is declared by using the keyword byte. It is used when sending data to and receiving data from a file or across a network. Choose a ‘byte’ whenever you need to move data to and from a file or across a network.

* **short ADT**(Java reserves 16 bits of main memory)

–It is ideal for use in programs that run on 16-bit computers and is the least used integer ADT. Choose a ‘short’ if you ever need to store an integer in a program that runs on a very old computer.

* **int ADT**(Java reserves 32 bits of main memory)

–It is the most frequently used ADT of the integer group for a number of reasons. Choose an ‘int’: for control variables in control loops in array indexes when performing integer math

* **long ADT**(Java reserves 64-bits of main memory)

–It is used whenever using whole numbers are beyond the range of an int data type.

**ADT – Floating Point**

**Floating-point** – stores real numbers (fractional values). A *real number* contains a decimal value. The precision of a number is the number of places after the decimal point that contains an accurate value.

* **float ADT**(Java reserves 32-bits of main memory)

–It used for real numbers that require single precision. Single precision means the value is precise up to 7 digits to the right of the decimal. Choose a ‘float’ whenever you need to store a decimal value where only 7 digits to the right of the decimal must be accurate.

* **double ADT**(Java reserves 64-bits of main memory)

–It is used to store real numbers that are very large or very small and require double the amount of memory that is reserved with a float ADT. Choose a ‘double’ whenever you need to store a decimal value where more than 7 digits to the right of the decimal must be accurate.

**ADT – Character**

* **char ADT**(Java reserves 16-bits of main memory)

–It is represented as an integer value that corresponds to a character set. A character set assigns an integer value to each character, punctuation, and symbol used in a language.

* + - Ex: the letter ‘A’ is stored in memory as the value 65, which corresponds to the letter ‘A’ in a character set.

–The keyword ‘char’ tells the computer that the integer stored in that memory location is treated as a character and not a number. There are two types of character sets:

* ASCII (American Standard Code for Information Interchange) uses a byte to represent a maximum of 256 characters of a language (English)
* Unicode

–used for languages (Russian, Arabic, Japanese, Chinese) that have more than 256 characters. It uses 2 bytes to represent each character.

NOTE: Choose a ‘char’ whenever you need to store a single character in memory.

**ADT – Boolean**

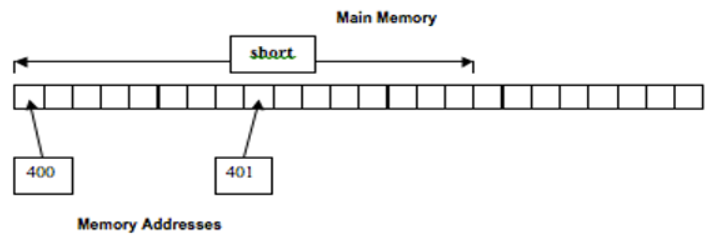
**Boolean**– stores a true or false value. The correct choice for storing a yes or no or true or false response to a question.

* **Boolean ADT**(Java reserves 1-bit of main memory)

–It reserves memory to store a ‘boolean’ value (true or false) represented as a zero or one. Choose a ‘boolean’ whenever you need to store one of two possibilities in memory.

#### ****Memory Addresses****

Imagine the main memory as a series of endless boxes organized into groups of eight. Each box holds a zero or one. Each group of eight boxes (1 byte) is assigned a unique number called a memory address.



#### ****How do Memory Address works?****

A memory address is indirectly or directly used within a program to access all eight boxes. The program tells the computer that it wants to copy data stored in memory location 401 – that is, the box whose address is 401. The computer goes to that memory location and copies the data (zero or one) from box 401 and copies data from the next seven boxes. Those next seven boxes do not have a memory address. They share the memory address of box 401.

#### ****Abstract Data Types and Memory Addresses****

Some ADT reserve memory in a size that is greater than 1 byte, for example, Java reserves 2 bytes of memory for a ‘short’ ADT (16-bits of main memory).

#### ****Algorithm****

An algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

**Basic Categories of Algorithm**

* **Search**− Algorithm to search an item in a data structure.
* **Sort**− Algorithm to sort items in a certain order.
* **Insert**− Algorithm to insert an item in a data structure.
* **Update**− Algorithm to update an existing item in a data structure.
* **Delete**− Algorithm to delete an existing item from a data structure.

**Characteristics of an Algorithm**

Not all procedures can be called an algorithm. An algorithm should have the following characteristics.

* **Unambiguous**− The algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
* **Input**− An algorithm should have 0 or more well-defined inputs.
* **Output**− An algorithm should have 1 or more well-defined outputs and should match the desired output.
* **Finiteness**− Algorithms must terminate after a finite number of steps.
* **Feasibility**− Should be feasible with the available resources.
* **Independent**− An algorithm should have step-by-step directions, which should be independent of any programming code.

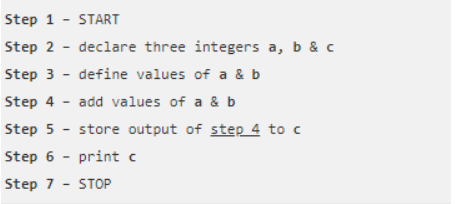
#### ****How to Write an Algorithm?****

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.

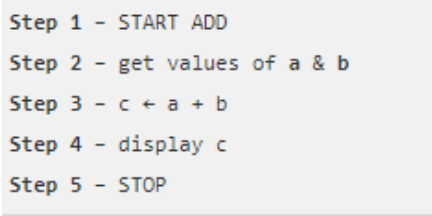
As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

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.

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



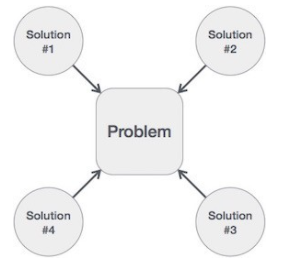
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –



In the 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. Observe what operations are being used and how the process is flowing.

Writing **step numbers** is optional.

Design an algorithm to get a solution to a given problem. A problem can be solved in more than one way.



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.

**Algorithm Analysis**

The efficiency of an algorithm can be analyzed at two different stages, before implementation, and after implementation. They are the following

* + **A Priori Analysis**− This is a theoretical analysis of an Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
  + **A Posterior Analysis**− This is an empirical analysis of an algorithm. The selected algorithm is implemented using a programming language. This is then executed on the target computer In this analysis, actual statistics like running time and space required, are collected.

#### ****Algorithm Complexity****

Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

**Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting

**Space Factor** − Space is measured by counting the maximum memory space required by the

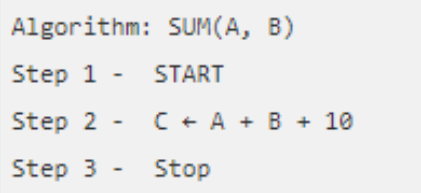
The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

**Space Complexity**

The space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

* A fixed part is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size,
* A variable part is a space required by variables, whose size depends on the size of the, for example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept



Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

#### ****Time Complexity****

The time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, the addition of two n-bit integers takes **n**steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

**Asymptotic analysis**

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst-case scenario of an algorithm.

 Asymptotic analysis is input bound i.e. if there's no input to the algorithm, it is concluded to work in constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n), and maybe for another operation, it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n**and the running time of the second operation will increase exponentially when **n**increases. Similarly, the running time of both operations will be nearly the same if **n**is significantly small.

The time required by an algorithm falls under three types −

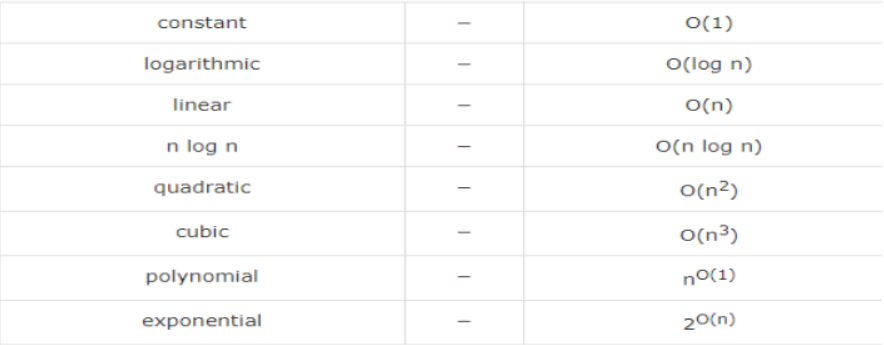
* **Best Case**− Minimum time required for program
* **Average Case**− Average time required for program
* **Worst Case**− Maximum time required for program

#### The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

#### ****Common Asymptotic Notation****



#### ****Asymptotic Notation****

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm—and that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, among other factors.  
  
Let's think about the running time of an algorithm more carefully. We can use a combination of two ideas. First, we need to determine how long the algorithm takes, in terms of the size of its input. This idea makes intuitive sense, doesn't it? We've already seen that the maximum number of guesses in linear search and binary search increases as the length of the array increases. Or think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So, we think about the running time of the algorithm as a function of the size of its input.

ArrayList

#### ****ArrayList****

ArrayList is a part of collection framework and is present in java.util package.  It provides a dynamic way of manipulating data.  Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.  Some of the notable characteristics of ArrayList are the following:

* ArrayList inherits AbstractList class and implements List interface.
* ArrayList is initialized by size, however, the size can increase if collection grows or shrunk if objects are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* Use a wrapper class if an ArrayList can not use primitive types, like int, char, etc.
* ArrayList in Java can be seen as similar to a vector in C++.

#### ****ArrayList Constructors****

1. **ArrayList()**: This constructor is used to build an empty array list
2. **ArrayList(Collection c)**: This constructor is used to build an array list initialized with the elements from collection c
3. **ArrayList(int capacity)**: This constructor is used to build an array list with initial capacity being specified

#### ****Basic Structure of an ArrayList****

ArrayList<Integer> arrayList1 = new ArrayList<Integer>();  
  
ArrayList<String> listarrayList2=new ArrayList<String>();  
  
ArrayList<Boolean> arrayList3 = new ArrayList<Boolean>();

#### ****ArrayList Methods****

|  |  |
| --- | --- |
| add(int index, Object element) | This method is used to insert a specific element at a specific position index in a list. |
| add(Object o) | This method is used to append a specific element to the end of a list. |
| addAll(Collection C) | This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection’s iterator. |
| addAll(int index, Collection C) | Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list. |
| clear() | This method is used to remove all the elements from any list. |
| clone() | This method is used to return a shallow copy of an ArrayList. |
| contains? (Object o) | Returns true if this list contains the specified element. |
| ensureCapacity?(int minCapacity) | Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| forEach?(Consumer<? super E> action) | Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| get?(int index) | Returns the element at the specified position in this list. |
| indexOf(Object O) | The index the first occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| isEmpty?() | Returns true if this list contains no elements. |
| lastIndexOf(Object O) | The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| listIterator?() | Returns a list iterator over the elements in this list (in proper sequence). |
| listIterator?(int index) | Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| remove?(int index) | Removes the element at the specified position in this list. |
| remove?(Object o) | Removes the first occurrence of the specified element from this list, if it is present. |
| removeAll?(Collection c) | Removes from this list all of its elements that are contained in the specified collection. |
| removeIf?(Predicate filter) | Removes all of the elements of this collection that satisfy the given predicate. |
| removeRange?(int fromIndex, int toIndex) | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| retainAll?(Collection<?> c) | Retains only the elements in this list that are contained in the specified collection. |
| set?(int index, E element) | Replaces the element at the specified position in this list with the specified element. |
| size?() | Returns the number of elements in this list. |
| spliterator?() | Creates a late-binding and fail-fast Spliterator over the elements in this list. |
| subList?(int fromIndex, int toIndex) | Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| toArray() | This method is used to return an array containing all of the elements in the list in the correct order. |
| toArray(Object[] O) | It is also used to return an array containing all of the elements in this list in the correct order same as the previous method. |
| trimToSize() | This method is used to trim the capacity of the instance of the ArrayList to the list’s current size. |

# Stack

**A stack** is a way to group things together by placing one thing on top of another and then removing things one at a time from the top of the stack.

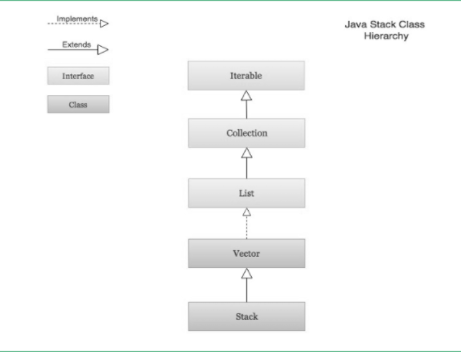
In computer science, a **stack** is **a Last-In, First-Out (L I F O)** abstract data type and data structure. A stack can have any abstract data type as an element but is characterized by only two fundamental operations: push and pop.

The **push**operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The **pop**operation removes an item from the top of the list and returns this value to the caller. A pop either reveals previously concealed items or results in an empty list.

#### ****Stack as Data Structure****

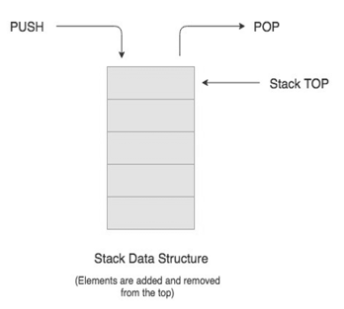
A **stack**is a restricted data structure because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.

A stack is a linear data structure that follows a particular order in which the operations are performed.  It is a part of Java’s collections framework. The following figure shows the class hierarchy of the Stack.



The Stack class extends Vector which implements the List interface.  A Vector is a re-sizable collection.  It grows its size to accommodate new elements and shrinks the size when the elements are removed.  Since the Stack class extends Vector; it also grows and shrinks its size as needed when new elements are added or removed.

#### ****Visualization of a Stack****



#### ****Real-life Example of a Stack****

Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

#### ****Creation of Stack****

Stack class is used by importing the **java.util library**.  A Stack object is created using the following structure.

Stack<BaseType> s = new Stack<BaseType>();

Just like ArrayList, Stack requires a base type to be an object

#### ****Stack Operations****

The **push( )** operation is used both to initialize the stack and to store values to it. It is responsible for inserting (copying) the value into the array and for incrementing the element counter (size). In a responsible implementation, it is also necessary to check whether the array is already full to prevent an overrun.

It is an operation used to insert or add a data item or element to the stack. The push ( ) method requires an item when called.  The push operation is used for insertion of new elements in the Stack.

The **pop( )** operation is responsible for removing a value from the stack, and decrementing the value of size. A responsible implementation will also need to check that the array is not already empty.

It is an operation used to delete or remove a data item or element at the top of the stack. The pop ( ) method returns the item being removed in the Stack.  The pop operation is used for the deletion of the top element of the Stack

The **size( )**operation is an operation to determine the size (number of items) of the Stack. It is used mainly to control loops. The size operation is used for checking the size of a Stack.

The **peek( )** operation is a method that looks at the item at the top of a stack. The peek ( ) method returns the item at the top without removing it.  The peek operation is used to determine what item is at the top of the Stack

The **search ( )** member method is a method that returns the position (in number) of an item from the top of a stack. The search ( ) method requires the desired item when called.  The search operation is used to determine the position of the item.

The **empty ( )** member method is a method that tests if a stack object is empty or not. The empty ( ) method returns either a boolean value of true or false. The method returns true if the stack is empty and false if it still contains an item or element.  The empty operation is used for checking if the Stack is empty.

#### ****Infix to Postfix Conversion Using Stack****

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine-readable form.  As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C\*D, D/A,** etc.

But in our usual form, an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)\*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

#### ****Infix Expression****

Infix expression follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>.  The expression of the form **a op b**.  When an operator is in-between every pair of operands.  Such an expression is termed infix expression, e.g., **A+B**.

#### ****Postfix Expression****

Postfix expression follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>.  The expression of the form **a b op**. When an operator is followed for every pair of an operand, e.g., **AB+**.

#### ****Why Postfix Representation of the Expression is Important?****

The compiler scans the expression either from left to right or from right to left.  Consider the below expression:

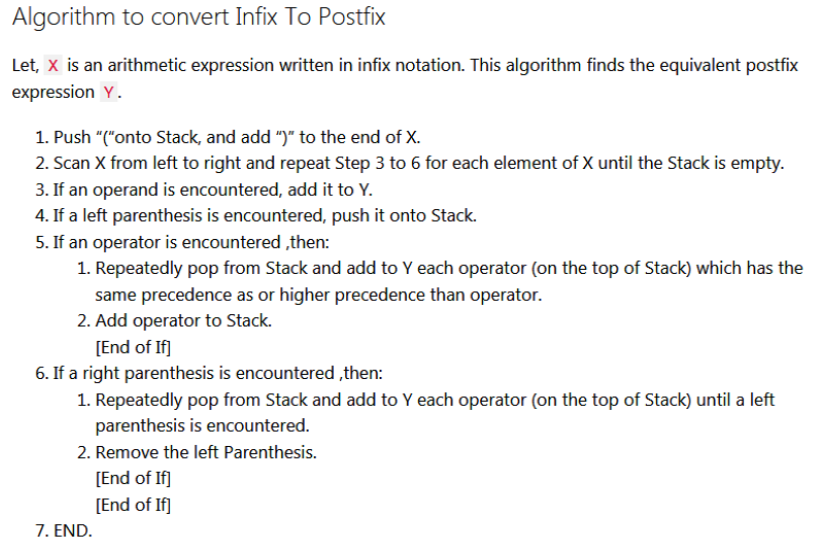
a op1 b op2 c op3 d  
  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression**b \* c**, then again scan the expression to add a to it. The result is then added to d after another scan.  The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.  The corresponding expression in the postfix form is **abc\*+d+**. The postfix expressions can be evaluated easily using a stack.

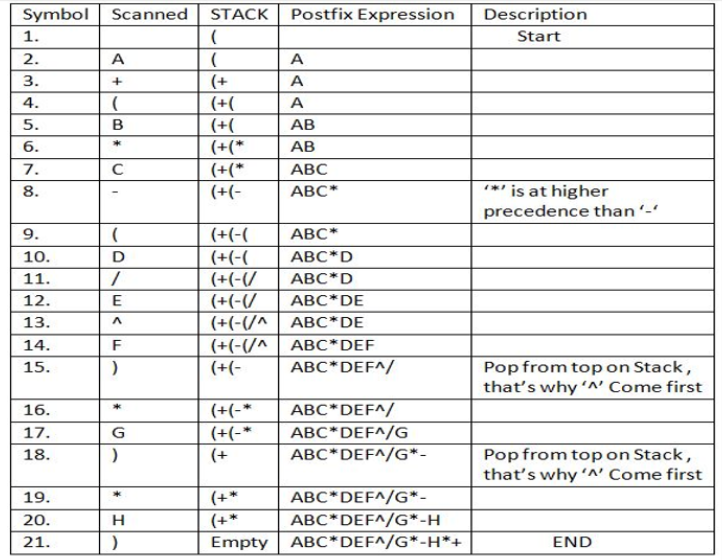
**Algorithm to convert Infix To Postfix**

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
   * If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.
   * Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

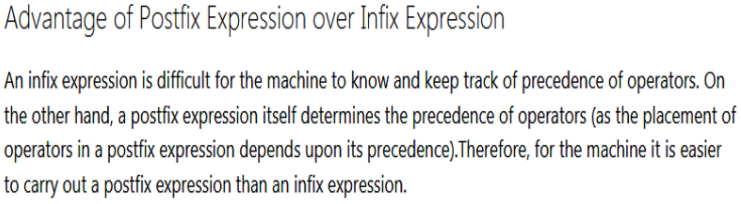
#### ****Algorithm to convert Infix To Postfix****



#### ****Conversion of Infix to Postfix****



Advantage of postfix expression over infix expression



**Prefix to Infix Conversion**

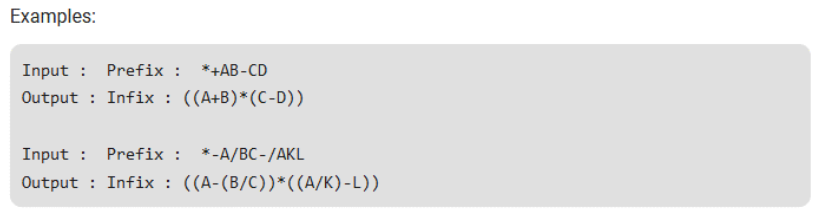
* **Infix**: An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

Example : (A+B) \* (C-D)

* **Prefix**: An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : \*+AB-CD (Infix : (A+B) \* (C-D) )

Given a Prefix expression, convert it into an Infix expression.  Computers usually do the computation in either prefix or postfix (usually postfix).  But for humans, its easier to understand an Infix expression rather than a prefix.  Hence conversion is needed for human understanding.



**Bubble Sort**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, the algorithm compares the first two elements and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

#### ****Bubble Sort Complexity****

Bubble sort is a simple sorting algorithm.  This sorting algorithm is a 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 Ο(n2) where **n** is the number of items.

#### ****Selection Sort****

The selection sort algorithm sorts an array by repeatedly finding the minimum element considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

1. The subarray is already sorted.
2. The remaining subarray is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

#### ****Selection Sort Complexity****

Selection sort 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 it is average and worst-case complexities are of Ο(n2), where **n** is the number of items.

#### ****Insertion Sort****

Insertion sort 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 that is to be inserted 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.

#### ****Insertion Sort Complexity****

This algorithm is not suitable for large data sets as its average and worst-case complexity are of Ο(n2), where n is the number of items

#### ****Shell Sort****

Shell sort is a highly efficient sorting algorithm and is based on the insertion sort algorithm.  This algorithm avoids large shifts as in the case of insertion sort if the smaller value is to the far right and has to be moved to the far left.  It uses insertion sort on widely spread elements, first to sort them and then sorts the less widely spaced elements. The spacing is termed as an **interval**.  The interval is calculated based on Knuth's formula as −



#### ****Shell Sort Complexity****

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items. And the worst-case space complexity is O(n).

#### ****Heap Sort****

**Heapsort** is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort.  Like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step.

#### ****Heap Sort Complexity****

Although somewhat slower in practice on most machines than quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime.

#### ****How Heap Sort Works****

Supposed { 6, 5, 3, 1, 8, 7, 2, 4 } are the elements of a list.  In building the heap, larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.

#### ****Merge Sort****

Merge Sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.  It uses a merge function to merge two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

#### ****Merge Sort Complexity****

The time complexity of Merge Sort is O(nLogn) in all 3 cases; worst, average, and best as merge sort always divides the array into two halves and takes linear time to merge two halves.

#### ****How the Merge Sort Works****

Merge sort organize array elements by recursively dividing the set into two halves until the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

**Quick Sort**

Quicksort is a divide and conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as a pivot.
2. Always pick the last element as the pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as a pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**Quick Sort Complexity**

*Worst Case:* The worst case occurs when the partition process always picks the greatest or smallest element as pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.

T(n) = T(0) + T(n-1) + (n) which is equivalent to T(n) = T(n-1) + (n)

= (n2).

**Best Case:** The best case occurs when the partition process always picks the middle element as a pivot. Following is recurrence for the best case.

T(n) = 2T(n/2) + (n)

= (nLogn)

**Average Case:**  
We can get an idea of average case by considering the case when partition, for instance,  O(n/9) elements in one set and O(9n/10) elements in another set.

T(n) = T(n/9) + T(9n/10) + (n)

= O(nLogn)

#### ****Bin/Bucket Sort****

Bin, also referred to as Bucket Sort runs in linear time on average. Like Counting Sort, bucket Sort is fast because it considers something about the input. Bucket Sort considers that the input is generated by a random process that distributes elements uniformly over the intervalμ=[0,1].

**Bin/Bucket Sort Algorithm**

1. Partition μ into n non-overlapping intervals called buckets.
2. Puts each input number into its buckets
3. Sort each bucket using a simple algorithm, e.g. Insertion Sort and then
4. Concatenate the sorted lists.

 Bucket Sort considers that the input is an n element array A and that each element A [i] in the array satisfies 0≤A [i] <1.

The code depends upon an auxiliary array B [0....n-1] of linked lists (buckets) and considers that there is a mechanism for maintaining such lists.

#### ****Radix Sort****

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order.  In the radix sort algorithm, a list of integer numbers is sorted based on the digits of individual numbers.  Sorting is performed from the least significant digit to the most significant digit.

Radix sort algorithm requires the number of passes that are equal to the number of digits present in the largest number among the list of numbers.

#### ****Queue****

 A queue is an n ordered list in which all insertions take place at one end, the**rear**, while all deletions take place at the other end, the **front**.  It is an example of a linear data structure.

A queue has a **First-In, First-Out (FIFO)**structure where elements can only be added to the rear of the queue and removed from the front of the queue.  It has two main operations, **enqueue** for insertion and **dequeue** for deletion.

#### ****Queue Restrictions****

The restrictions on queue imply that the first element which is inserted into the queue will be the first one to be removed.  Thus A is the first letter to be removed, and queues are known as **First In First Out (FIFO)** lists.

A **LinkedList** is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.  A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

There are three types of LinkedList:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

A **Singly Linked List** is the most common form of a Linked List where each node contains a data field and a **single** pointer to the next node in the list.

The reference to the **first** node in the list is called the **HEAD** of the list. The pointer/reference/link field contained in the node is used to traverse to the next node and to its next node and so on till we reach a node that points to NULL. This is the **last** node in the list.  Also, a singly linked list can only be traversed in one and only one direction i.e. from head to the last node. There is no way to traverse from the last node back to the head. The following is an example of a singly linked list with 5 nodes.

**Inserting Nodes Into a Singly Linked List**

There are three cases to consider for inserting a node into a singly linked list. Adding a node to the :

* Beginning of the list.
* End of the list.
* Specified position in the list.

**Inserting Nodes at the Beginning of the List**

To insert a new node at the beginning of the list the following algorithm is used :

* Assign the reference of the HEAD to the new node’s next field.
* Make the new node as the HEAD of the list.

**Inserting a Node at the End of the List**

To insert a node at the end of the list the following algorithm is used.

* Traverse the list until we find the last node.
* The new node’s reference is assigned to the last node’s next field.

**Inserting a Node at a Specified Position in the List**

To insert a node at a specified position in the list the following algorithm is used.

* Traverse (position – 1) times or till the end of the list is reached and maintain previous and current references.
* Assign the reference of the new node to the prev node’s next field.
* Assign the cur node’s reference to the new node’s next field.

**Deleting Nodes From a Singly Linked List**

Deleting a node from a singly linked list can be a little complex as the node to be deleted can be the first node, the last node, or a node in the middle of the list. Let us discuss each case.

* **First Node:** If the node to be deleted is the first node itself, we assign the reference of the next of the HEAD node to the HEAD node.
* **Last Node or Any Other Node:** To delete any other node in the list, we traverse the list keeping track of the previous and current nodes in the list until we find the node to be deleted with the required data field or we reach the end of the list i.e. NULL without finding the data element in the list.

If the node is found, we assign the reference of the next field of the current node to the previous node’s next.

#### ****Doubly Linked List****

A **Doubly Linked List** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

**Difference Between Singly Linked List and Doubly Linked List**

Both types of lists contain a pointer to the next node, as well as a data field to represent the actual value stored in the node.  The only difference between Doubly LinkedList and SinglyLinkedList is that the Doubly LinkedList also contains a pointer to the previous node, not just the next node.

A Doubly LinkedList must contain three variables:

* data variable
* next node variable
* previous node variable

#### ****Circular Linked List****

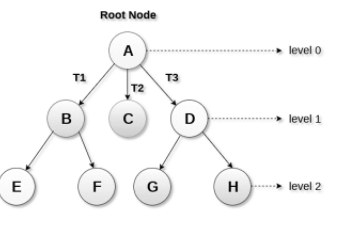
A**Circular Linked List** is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

A **tree** is a recursive data structure containing the set of one or more data nodes where one node is designated as the **root** of the tree while the remaining nodes are called as the **children** of the root.

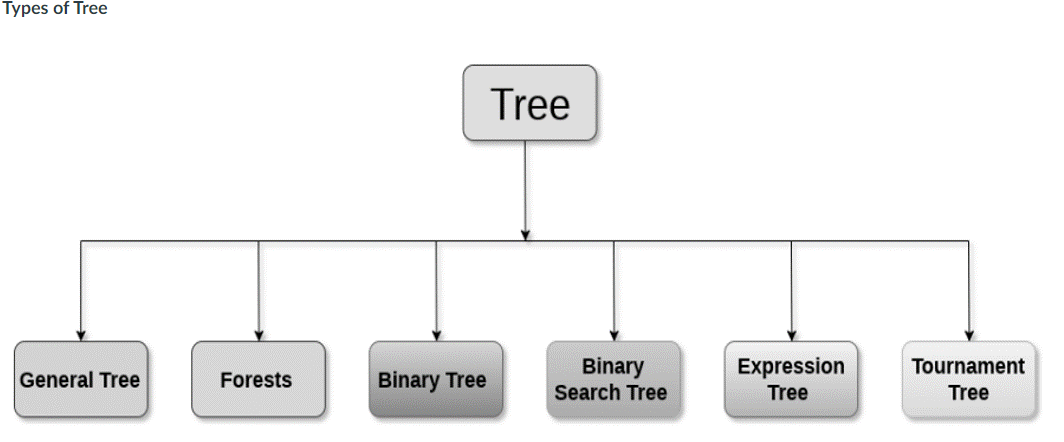
The nodes other than the root node are partitioned into the non-empty sets where each one of them is to be called **sub-tree.**

Nodes of a tree either maintain a **parent-child relationship**between them or they are **sister nodes**.

In a general tree, a node can have any number of children nodes but it can have only a single parent.

node A is the root node of the tree while the other nodes can be seen as the children of A. 

|  |  |
| --- | --- |
| Term | Description |
| **Root Node** | The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. |
| **Sub Tree** | If the root node is not null, the tree T1, T2, and T3 are called sub-trees of the root node. |
| **Leaf Node** | The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes. |
| **Path** | The sequence of consecutive edges is called a path. In the tree shown in the above image, the path to the node E is A→ B → E. |
| **Ancestor node** | An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F has the ancestors, B, and A. |
| **Degree**:- | The degree of a node is equal to a number of children, a node has. In the tree shown in the above image, the degree of node B is 2. The degree of a leaf node is always 0 while in a complete binary tree, the degree of each node is equal to 2. |
| **Level Number**:- | Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. The root node of the tree is always present at level 0. |



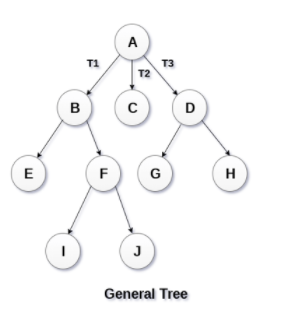
#### ****General Tree****

A **general tree** stores the elements in a hierarchical order in which the top-level element is always present at level 0 as the root element.

All the nodes except the root node are present a number of levels.

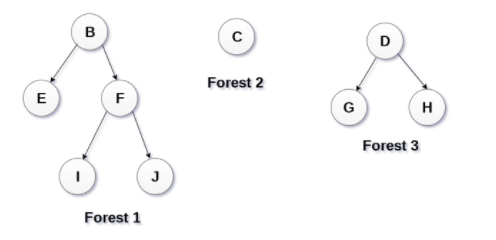
The nodes which are present on the same level are called siblings while the nodes which are present on the different levels exhibit the parent-child relationship among them.

A node may contain any number of sub-trees.  The tree in which each node contains 3 sub-tree, is called a **ternary**tree.



#### ****Forest****

Forest can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connect the root node to the first-level node.



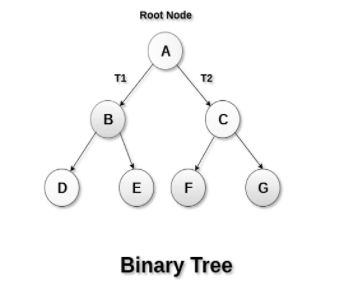
**Binary Tree**

A **binary tree** is a data structure in which each node can have at most 2 children.

* The node present at the topmost level is called the root node.
* A node with the 0 children is called the leaf node.
* Binary Trees are used in applications like expression evaluation and many more.

**Binary Tree** is a special type of generic tree in which, each node can have at most two children.  A binary tree is generally partitioned into three disjoint subsets.

* The root of the node
* left sub-tree which is also a binary tree.
* Right binary sub-tree



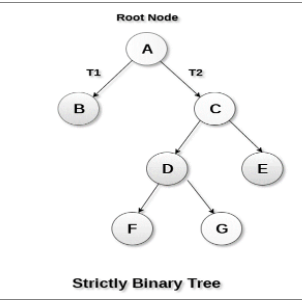
**Types of Binary Tree**

1. Strictly Binary Tree
2. Complete Binary Tree

In Strictly Binary Tree, every non-leaf node contains non-empty left and right sub-trees.

In other words, the degree of every non-leaf node will always be 2.

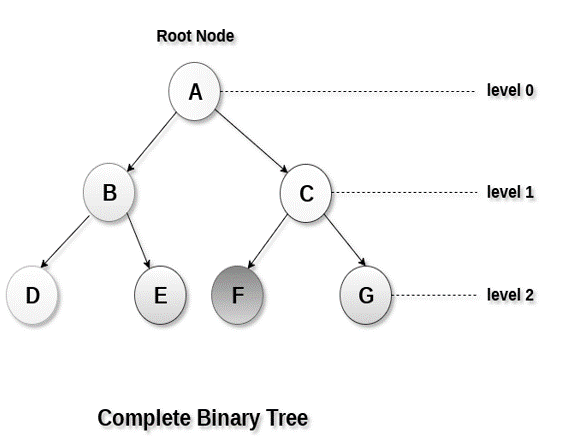
A strictly binary tree with n leaves will have (2n - 1) nodes.



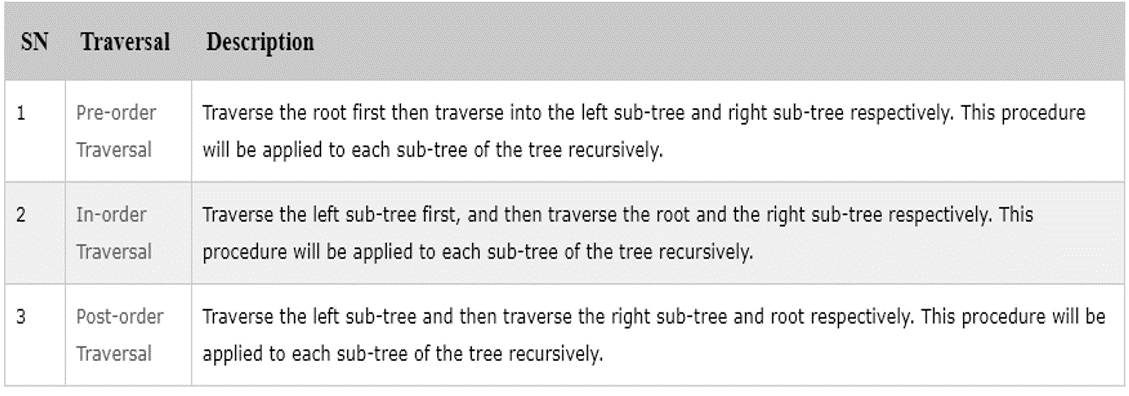
A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d.

A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d.

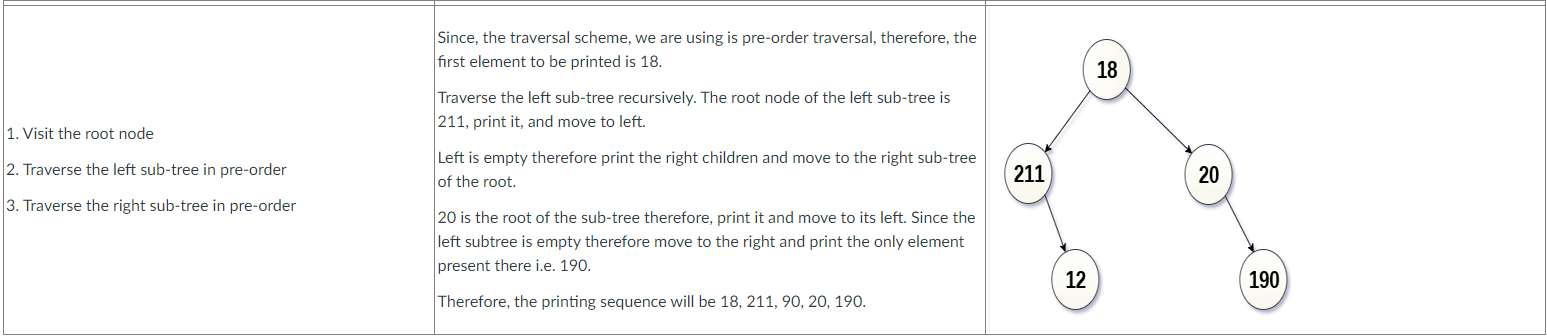
The total number of nodes in a complete binary tree with depth d is 2d+1-1 where leaf nodes are 2dwhile non-leaf nodes are 2d-1.



#### ****Binary Tree Traversal****



#### ****Binary Pre-order Traversal****

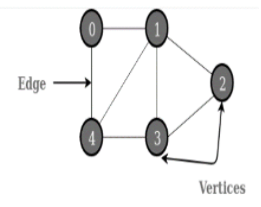


**Graph**

A **graph** is a non-linear data structure consisting of **nodes** and **edges**.  The nodes are sometimes also referred to as **vertices**.  The edges are lines or arcs that connect any two nodes in the graph.  The graph consists of a finite set of vertices(or nodes) and set of edges which connect a pair of nodes

In the graph, the

* set of vertices are **V = {0,1,2,3,4} ,**and the
* set set of edges are  **E = {01, 12, 23, 34, 04, 14, 13}.**



**Application of Graph**

Graph are used to solve many real-life problems.

1. It is used to represent networks.  The networks may include paths in a city or telephone network or circuit network.
2. It is used in social networks like LinkedIn, Facebook.  For example, on Facebook, each person is represented with a vertex(or node).  Each node is a structure and contains information like person id, name, gender, locale, etc.

**Graph and its Representations**

A graph is a data structure that consists of the following components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pairs of the form (u, v) called the edge.

The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph (digraph).

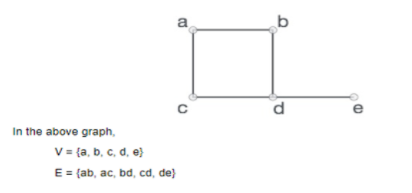
The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.

The edges may contain weight/value/cost.

#### ****Graph Data Structure****

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 **E**is the set of edges, connecting the pairs of vertices.



#### ****Graph Data Structure Important Terms****

|  |  |
| --- | --- |
| **Vertex** | Each node of the graph is represented as a vertex. |
| **Edge** | An edge represents a path between two vertices or a line between two vertices. |
| **Adjacency** | Two nodes or vertices are adjacent if they are connected to each other through an edge. |
| **Path** | Represents a sequence of edges between the two vertices. |

**Graph Data Structure Basic Operations**

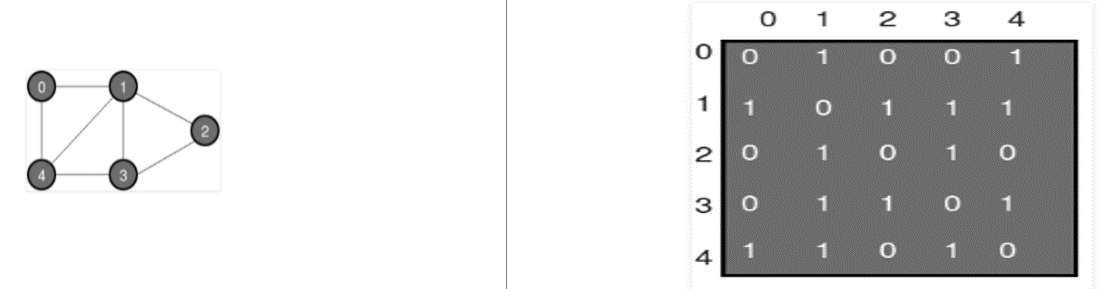
* **Add Vertex.**Adds a vertex to the graph.
* **Add Edge.**Adds an edge between the two vertices of the graph.
* **Display Vertex.**Displays a vertex of the graph.

**Graph Representations**

1. Adjacency Matrix
2. Adjacency List

#### ****Adjacency Matrix****

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.  Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.  The adjacency matrix for the undirected graph is always symmetric.  Adjacency Matrix is also used to represent weighted graphs.  If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



**Pros and Cons of Adjacency Matrix**

**Pros**

* Representation is easier to implement and follow.
* Removing an edge takes O(1) time.
* Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

**Cons**

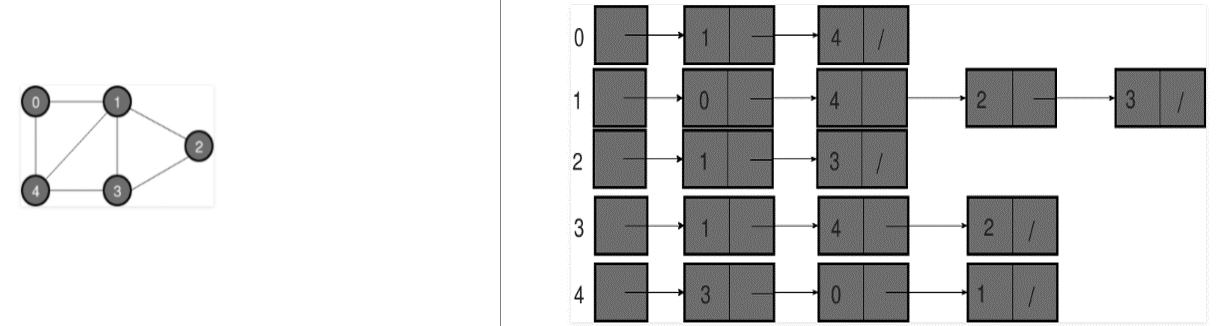
* Consumes more space O(V^2).
* Even if the graph is sparse(contains less number of edges), it consumes the same space.
* Adding a vertex is O(V^2) time.

**Adjacency List**

An array of lists is used in the adjacency list type of graph representation.  The size of the array is equal to the number of vertices.

Let the array be an array[].

* An entry array[i] represents the list of vertices adjacent to the***i***th vertex.
* The representation can also be used to represent a weighted graph.
* The weights of edges can be represented as lists of pairs.



#### ****Breadth-First Search or BFS for a Graph****

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.  The only catch here is, unlike trees, graphs may contain cycles, so you may come to the same node again.  To avoid processing a node more than once, use a boolean visited array.  For simplicity, it is assumed that all vertices are reachable from the starting vertex.

**Depth First Search or DFS for a Graph**

Depth First Traversal (or Search) for a graph is similar to the Depth First Traversal of a tree.  The only catch here is, unlike trees, graphs may contain cycles, so you may come to the same node again.  To avoid processing a node more than once, we use a boolean visited array.